VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Data Structures

UNIT-IV

Syllabus

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- **UNIT-4: Graphs:** Recap of Graphs, Connected Components, Bi connected components,
- **Minimum Cost Spanning Tree**: Prims Algorithm (recap), Kruskal's Algorithm (recap), Sollin's (Boruvka's) Algortihm
- **Single Source Shortest Path**: Dijkstra's Algorithm, Bellman Ford Algorithm
- **Transitive Closure** : Warshall's Algorithm
- **All Pairs Shortest Path**: Floyd's Algorithm

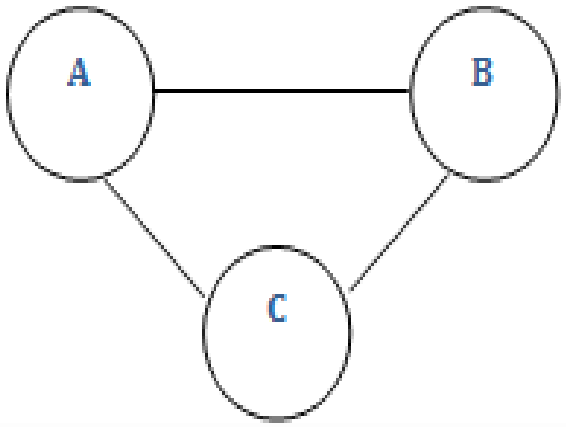
Graphs

- A Graph G consists of a set V of vertices (Nodes) and a set of edges (arcs)E. We write G = (V, E).
- V is a finite and non empty set of vertices.
- E is a set of pairs of vertices called edges.
- An edge e = (v, w) is a pair of vertices v and w, and is said to be incident with v and w. Nodes v and w are called as endpoints of e, and v and w are said to be adjacent nodes or neighbors.

Graphs

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Ex: An example graph G



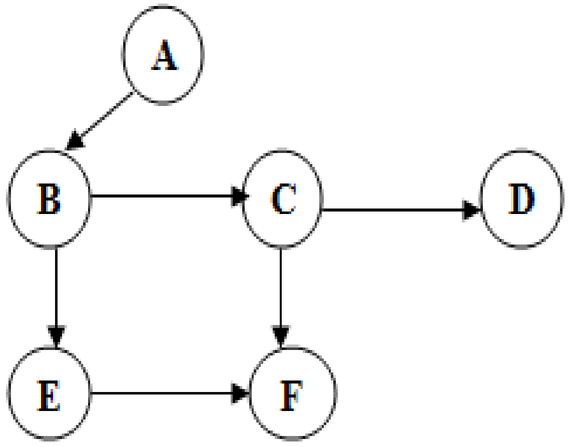
- V (G) = { A, B, C }
- E (G) = { (A, B) (A, C) (B, C) }

Graph VS Tree

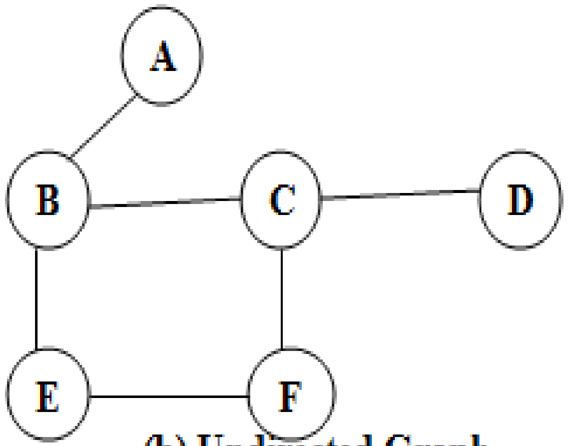
Graph	Tree
Non-linear data structure.	Non-linear data structure.
Collection of vertices and edges.	Collection of vertices and edges.
Each node can have any number of edges.	In binary trees every node can have at the most 2 child nodes.
There is no unique node like trees.	There is a unique node called root node.
A cycle can be formed.	There will not be any cycle.
Applications: used for finding shortest path in networking graph.	Applications: used for expression trees, and decision tree.

Types of Graphs

- **Directed graphs** have edges with direction.
- **Undirected graphs** have edges that do not have a direction.



(a) Directed Graph



(b) Undirected Graph

Graphs

- A **path** is a sequence of vertices in which each vertex is adjacent to the next one. In directed graph {A, B, C, F} is one path and {A, B, E, F} is another.
- Two vertices in a graph are said to be **adjacent** vertices (or) neighbors if there is a path of length 1 connecting them.
- A **cycle** is a path consisting of at least three vertices that starts and ends with the same vertex.
- A **loop** is a special case of a cycle in which a single arc begins and ends with the same vertex.

Representation of Graphs

- There are two representation methods for graphs.
  - 1. Adjacency Matrix
  - 2. Adjacency Lists
- **<u>Adjacency Matrix:</u>** The Adjacency Matrix A for a graph G = (V, E) with n vertices, is an n x n Matrix of bits, such that A

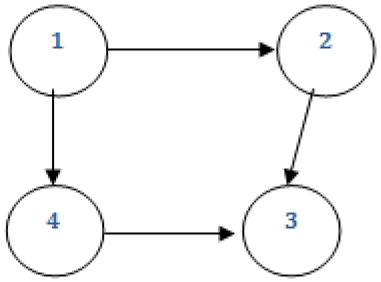
$$A_{ij} = \begin{cases} 1 & \text{If there is an edge } V_i \text{ to } V_j \text{(un weighted graph)} \\ 0 & \text{If not} \end{cases}$$

- In case of weighted graph we are measuring a weight instead of 1.

Representation of Graphs

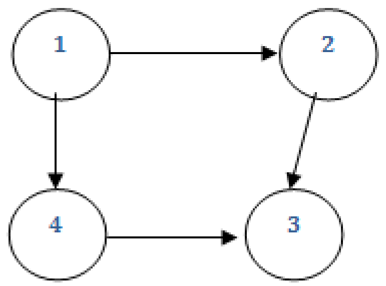
The Adjacency Matrix would be

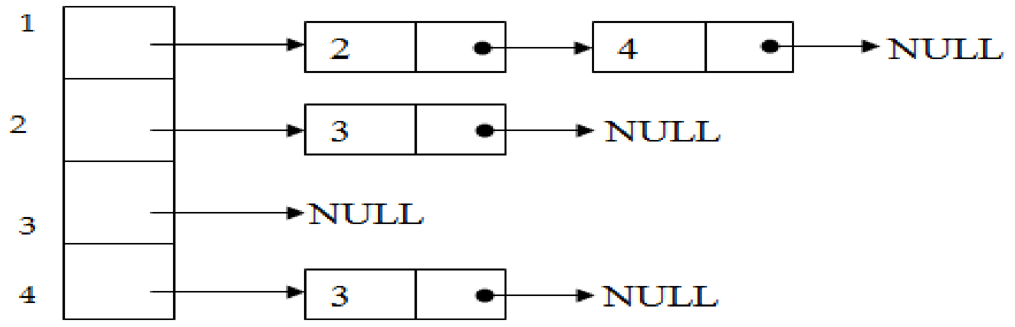
$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$



Representation of Graphs

- Adjacency List Representation: In this representation, we store a graph as a linked structure. We store all the vertices in a list and then for each vertex, we have a linked list of the adjacent vertices.
- Ex: Let us consider a Graph





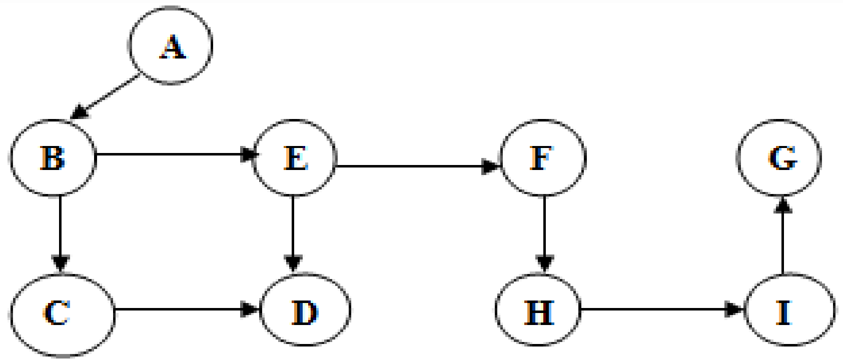
Operations on Graphs

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

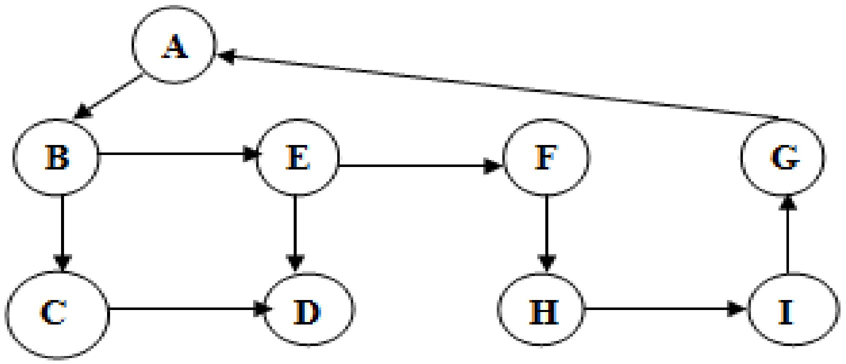
- Insert vertex.
- Delete vertex.
- Insert edge.
- Delete edge.
- Find vertex.

Types of Graphs

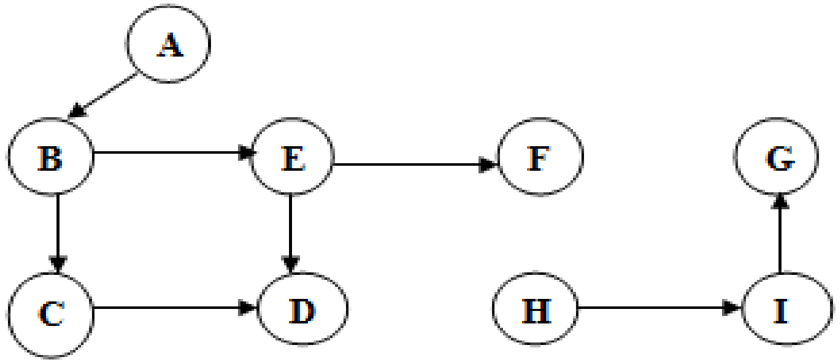
- A directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph.
- A directed graph is **weakly connected** if at least two vertices are not connected.
- A graph is **disconnected** if at least two vertices of the graph are not connected by a path. If a graph G is disconnected, then every maximal connected subgraph of G is called a connected component of the graph G.



(d) Weakly Connected



(e) Strongly Connected

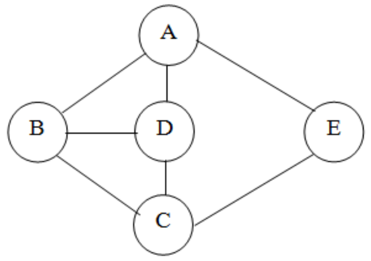


(f) Disjoint Graph

Graph Traversals

Depth first Search

```
Void DFS(Vertex V)
    {
        visited of all vertices initially false
        push v to stack;
        while stack not empty
                v=pop();
                if (!visited[v])
                        visited[v]=true;
                for each Vertex W adjacent to V
                        if(!Visited[W])
                        push(W);
    }
```

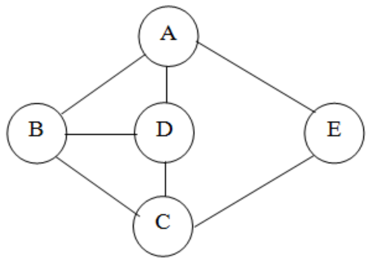


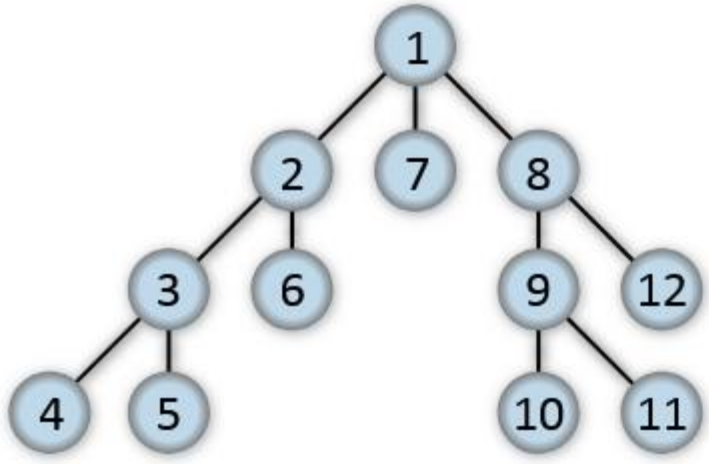
Graph Traversals

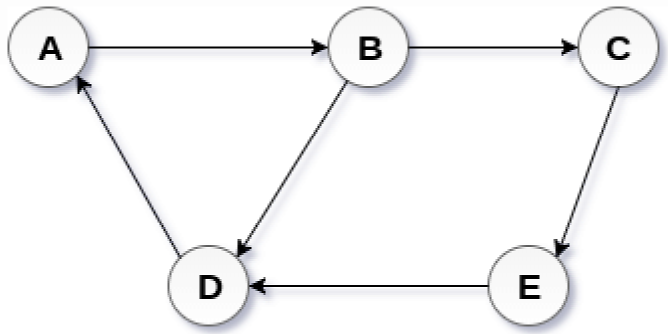
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Breadth first Search

```
Void BFS(Vertex V)
    {
        visited of all vertices initially false
        initialize Q to be Empty //Q is a Queue
        Enqueue(V);
        visited[V]=true;
        while(Q not empty)
         {
            V=Dequeue()
            for each vertex W adjacent to V
                if(visited[W] = false)
                 {
                    Enqueue( W);
                    Visited[W] = True;
                 } }
```

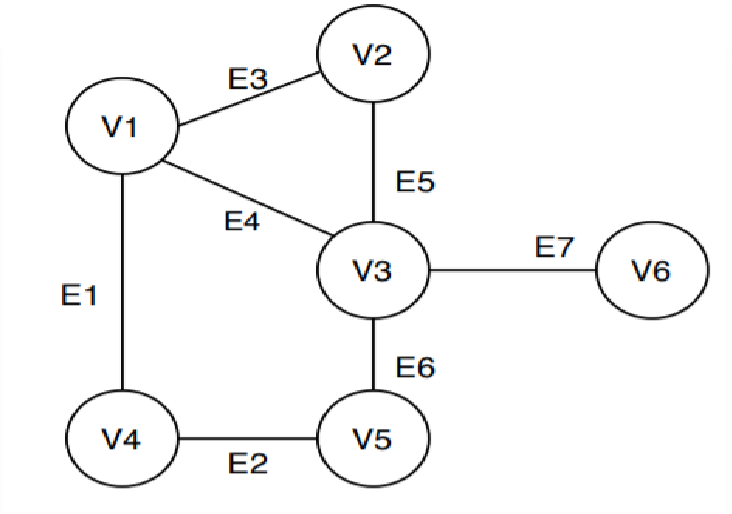


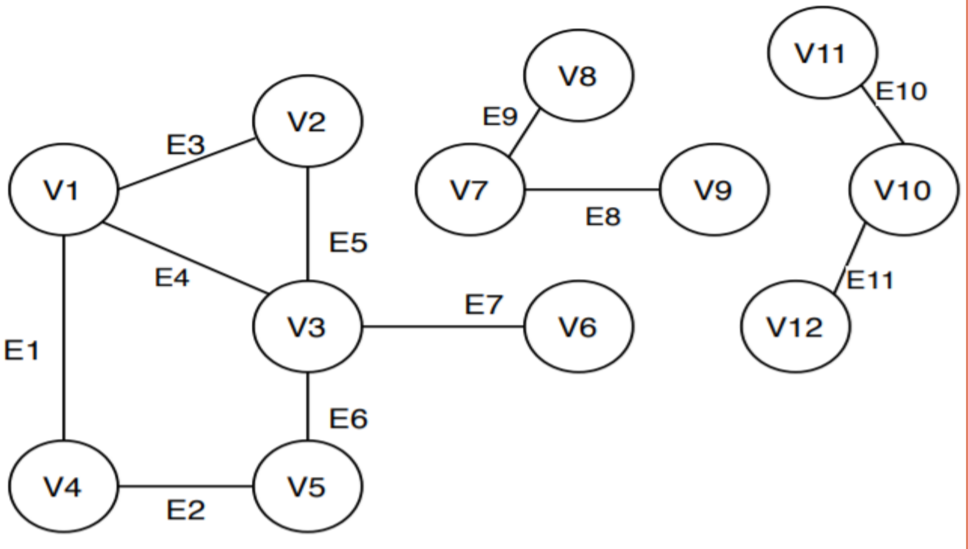

1
2
7
8
3
6
9
12
4
5
10
11


A
B
C
D
E

Connected Components

- A connected component or simply component of an undirected graph is a subgrah in which each pair of nodes is connected with each other via a path.
- A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges.



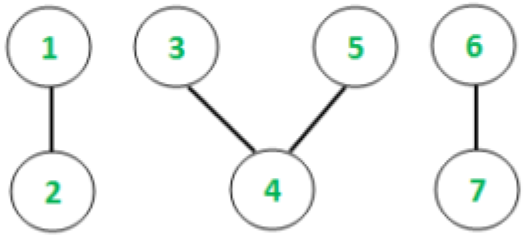


Connected Components

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Algorithm 1: Finding Connected Components using DFS

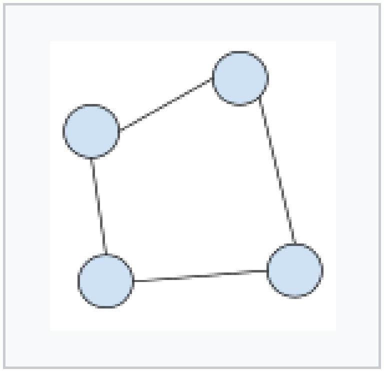
```
Data: Given an undirected graph G(V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex $k \in V$ do
    Visited[k] = False;
end
for each vertex $k \in V$ do
    if Visited[k] == False then
        DFS(V,k);
        Component_Count = Component_Count + 1;
    end
end
Print Component_Count;
Procedure DFS(V,k)
Visited[k] = True;
for each vertex $p \in V.Adj[k]$ do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

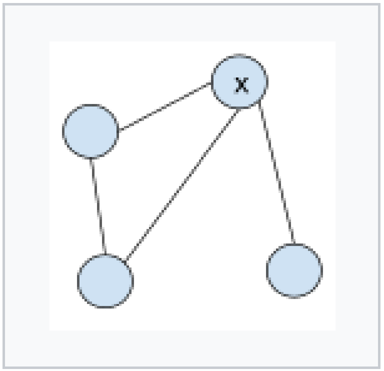


BiConnected Components

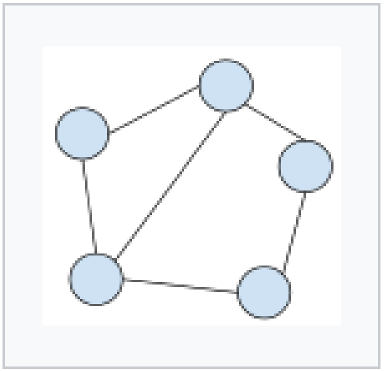
3A15

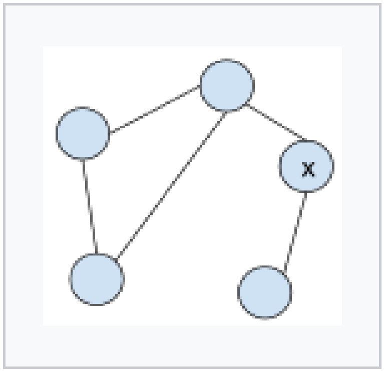
- A graph is said to be Biconnected if It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path, even after removing any vertex the graph remains connected.


A biconnected graph on four vertices and four edges



A graph that is not biconnected. The removal of vertex x would disconnect the graph.

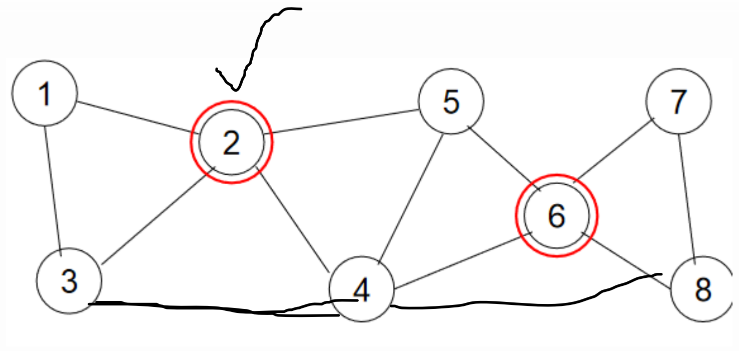

A biconnected graph on five vertices and six edges



A graph that is not biconnected. The removal of vertex x would disconnect the graph.

Articulation Points

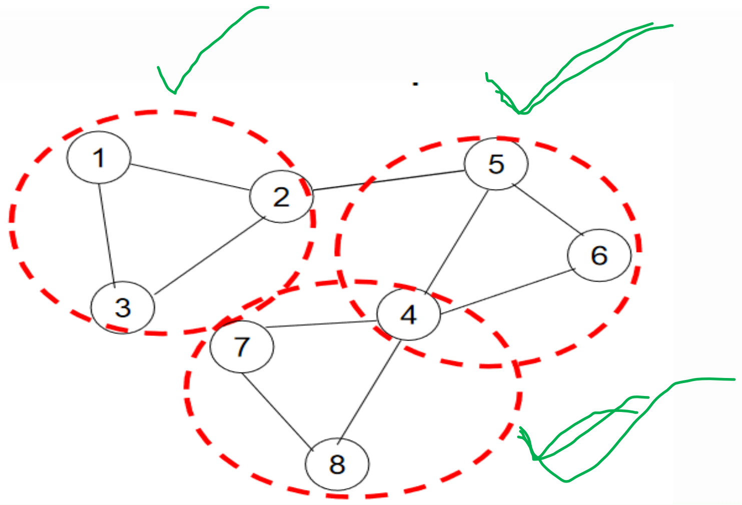
- An articulation point of a graph is a vertex v such that the removal of v will not break the graph into two or more pieces.
- A connected graph with no articulation points is said to be biconnected.



BiConnected Components

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- *Biconnected components:* A biconnected component of a graph is a maximal biconnected subgraph. Two bi-coonected componets can have one vertex in common that point is articulation point. Hence no edge can be in two bi-connected components.
- Depth-first search is particularly useful in finding the biconnected components of a graph.



Algorithm for finding Articulation Points

[illegible]

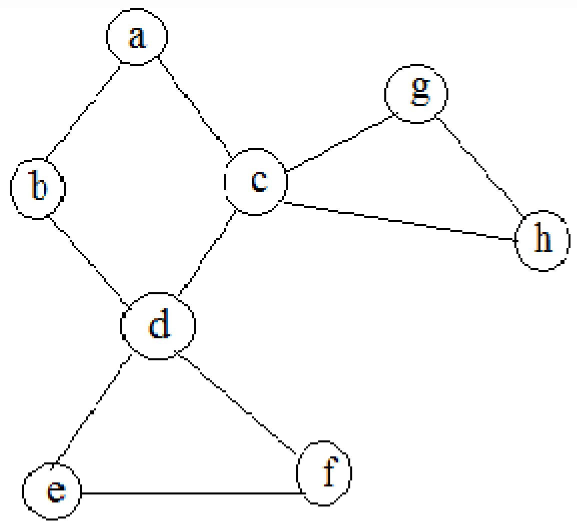
- **Step 1**: Find DFS and assign discovery number(dfn)to every vertex.
- **Step 2**: Find Low() for every vertex.
- ✓ **Step 3**: Check if dfn(u)<=Low(v)[u parent, v child] then u is articulation point.

dfn: *The number at which the vertex is discovered in DFS*

BackEdges: *are the edges which are present in the original graph but not present in the DFS Spanning tree*

Low(): *The lowest discovery number that can be reached from the current vertext by taking one back edge*

✓ *Low(u)=min(dfn of u, min of lowest value of children,min of dfn value of back edges)*



Vertex	h	g	c					
dfn	8	7	6					
Low	6	6	1					

Spanning Tree

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

SPANNING TREES:

- A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees.

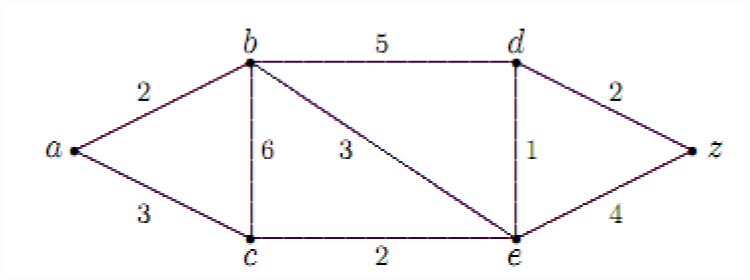
Minimum cost spanning tree

- Minimum cost spanning tree of a weight connected graph is a spanning tree with minimum cost or weight.
- There are three ways to find minimum cost spanning trees:
    - Prim's algorithm
    - Kruskal's algorithm
    - Sollin's algorithm((Boruvka's)

Prim’s Algorithm

- **procedurePrim(G: weighted connected graph with n vertices)**

```
T := a minimum-weight edge
fori = 1 to n − 2
begin
  e:= an edge of minimum weight adjacent to a vertex in T   and not forming a cycle in T if added to T
  T :=T with e added
end
return(T).
```



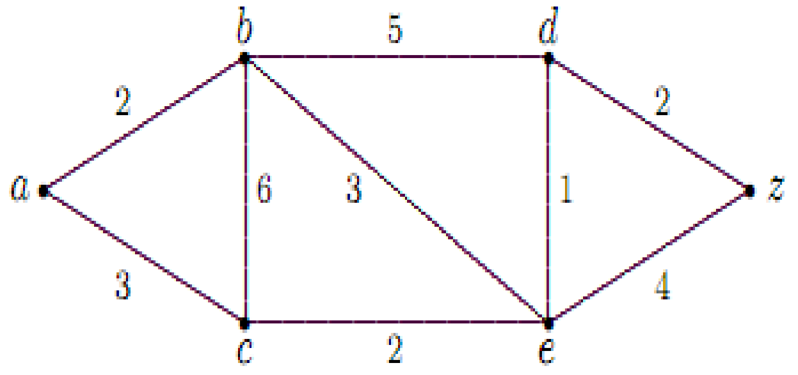
VASIREDDY VENKATADRI
INSTITUTE OF TECHNOLOGY

Kruskal's Algorithm

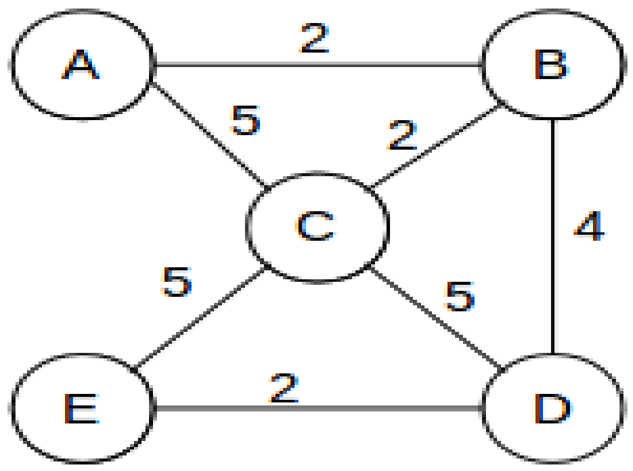
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

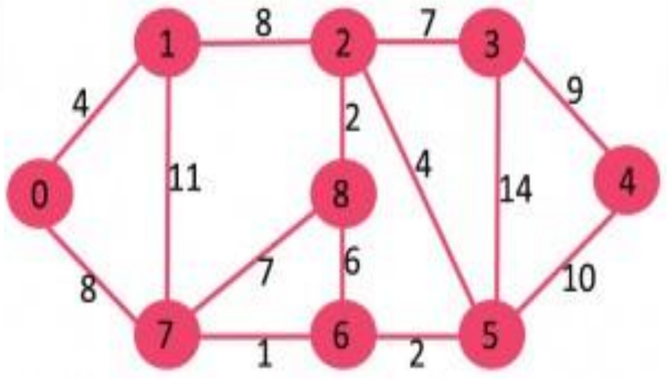
Kruskal's algorithm:

- sort the edges of G in increasing order by length
- keep a subgraph S of G, initially empty
- for each edge e in sorted order
- if the endpoints of e are disconnected in S
- add e to S
- return S



Kruskal's Algorithm



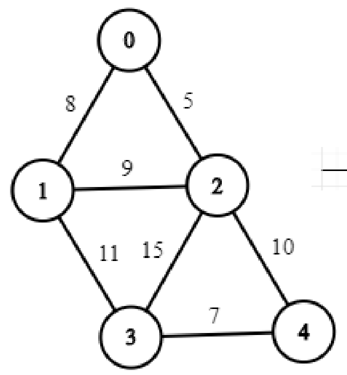


Sollin's(Boruvka's) Algorithm

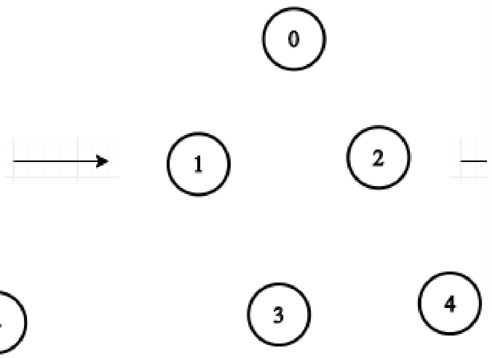
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- 1) Input is a connected, weighted and un-directed graph.
- 2) Initialize all vertices as individual components (or sets).
- 3) Initialize MST as empty.
- 4) While there are more than one components, do following for each component.
  - a) Find the closest weight edge that connects this component to any other component.
  - b) Add this closest edge to MST if not already added.
- 5) Return MST.

Sollin's(Boruka's) Algorithm

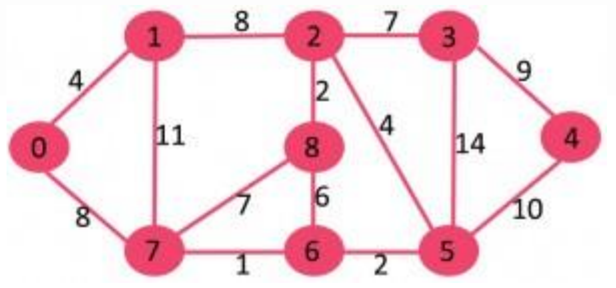


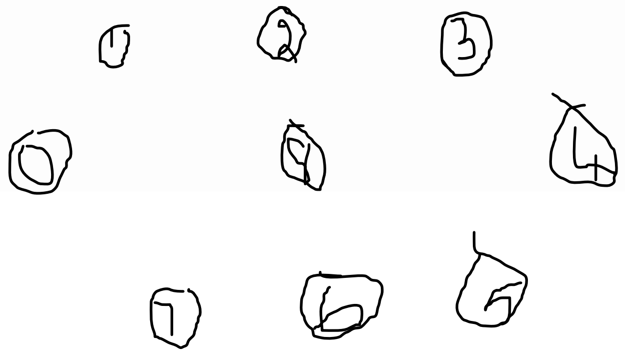
Step 0: The Graph

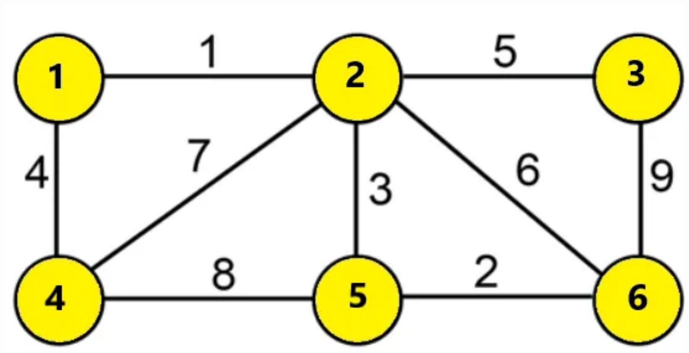


Step 1: Bunch of unconnected trees

Sollin's(Boruka’s) Algorithm

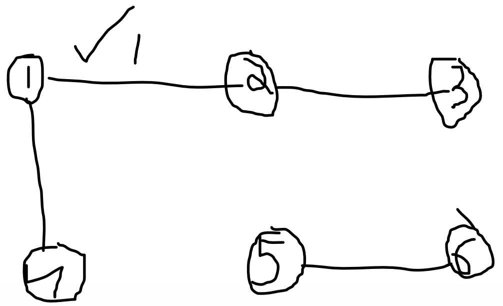






min

1	2	3	4	5	6
1	1	6	2	8	9



Single-source shortest path

For a weighted graph G=(V,E,w), the single source shortest path problem is to find the shortest paths from a vertex v € V to all other vertices in V.

There are two algorithm to solve this problem.

- Dijktra's algorithm
- Belleman Ford algorithm

Dijktra's algorithm

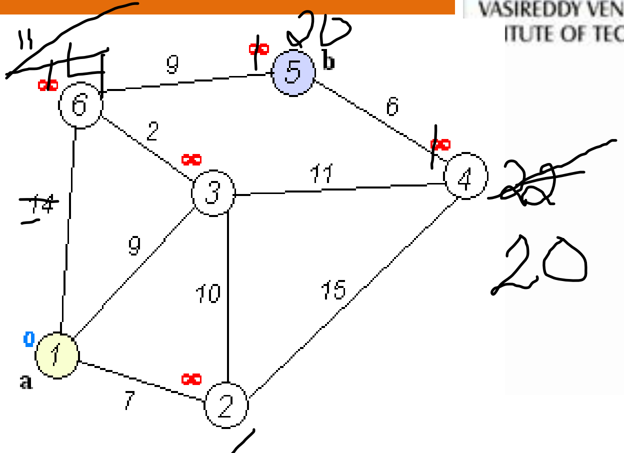
```
function Dijkstra(Graph, source):
2    for each vertex v in Graph:
3        dist[v] := infinity ;
4        previous[v] := undefined ;
5    end for ;
6    dist[source] := 0 ;
7    Q := the set of all nodes in Graph ;
8    while Q is not empty
9        u := vertex in Q with smallest distance in dist[] ;
10        If dist[u] = infinity:
11        break ;
12            end if ;
13        remove u from Q ;
14        for each neighbor v of u:
15                alt := dist[u] + dist_between(u, v) ;
16            if alt < dist[v]:
17                dist[v] := alt ;
18                previous[v] := u ;
19                decrease-key v in Q;
20            end if ;
21        end for ;
22    end while ;
23    Return dist[] ;
24  end Dijkstra.
```

$u = 4$

$v \to 5$

$20 + 6 = 26$

P



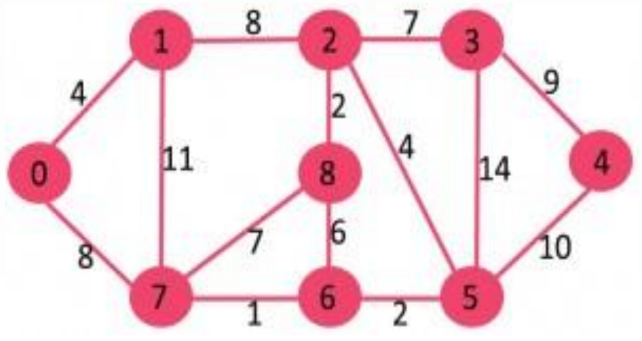
1	2	3	4	5	6
0	7	9	20	20	11

1	2	3	4	5	6
-1	1	1	3	6	3

If we are only interested in a shortest path between vertices *source* and *target*, we can terminate the search at line 13 if u = *target*. Now we can read the shortest path from *source* to *target* by iteration:

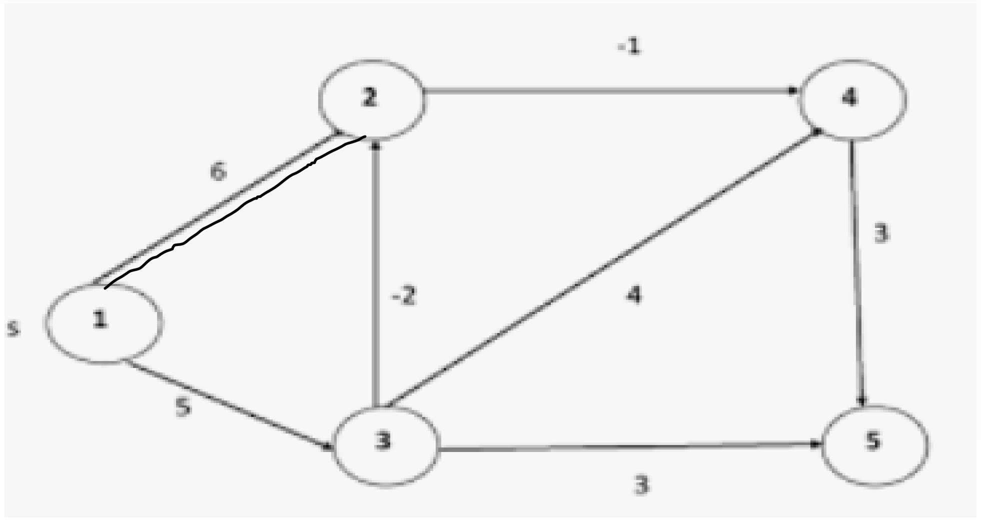
```
1  S := empty sequence
2  u := target
3  while previous[u] is defined:
4      insert u at the beginning of S
5      u := previous[u]
6  end while ;
```

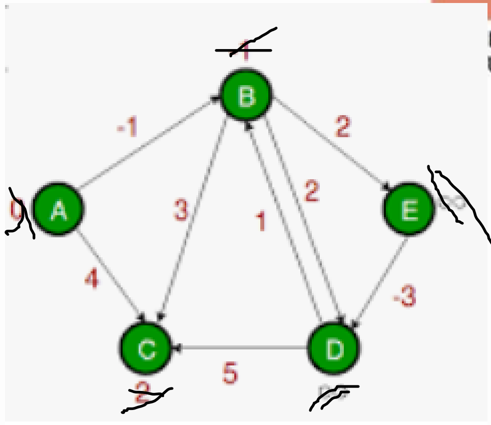
Dijktra’s algorithm



BellmanFord Algorithm

- **function** BellmanFord(*list* vertices, *list* edges, *vertex* source) **is**
- distance := *list* of size *n*
- predecessor := *list* of size *n*
- **for each** vertex v **in** vertices **do**
  - distance[v] := **inf** predecessor[v] := null
- distance[source] := 0
- **repeat** |V|−1 **times**:
  - **for each** edge (u, v) **with** weight w **in** edges **do**
    - **if** distance[u] + w < distance[v] **then**
    - distance[v] := distance[u] + w
    - predecessor[v] := u
- **for each** edge (u, v) **with** weight w **in** edges **do**
- **if** distance[u] + w < distance[v] **then error** "Graph contains a negative-weight cycle"
- **return** distance, predecessor





A	B	C	D	E		
1	2	3	4	5	6	7
0	-1	2	-2	1		

(1 2)

0, -1, 1

[illegible]

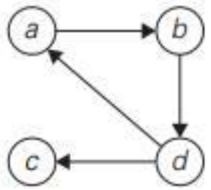
1	2	3	4	5	6	7

Transitive closure

- Consider a directed graph G=(V,E), where V is the set of vertices and E is the set of edges. The transitive closure of G is a graph G+ = (V,E+) such that for all v,w in V there is an edge (v,w) in E+ if and only if there is a non-null path from v to w in G.
- We have seen Adjacency Matrix, which tells you whose directly connected to whom, but there can be many other paths which may exist and might not be visible from the Adjacency matrix, as these paths may have intermediate vertices. To find all such round about or via-paths there is a procedure to be followed, known as the transitive closure, which results in a matrix showing all those via-paths, if ever exist.
- We can find the transitive closure using Floyd Warshall Algorithm

Floyd Warshall Algorithm

```
transColsure(graph)
Input: The given graph.
Output: Transitive Closure matrix.
Begin
 copy the adjacency matrix into another matrix named transMat
 for any vertex k in the graph, do
 for each vertex i in the graph, do
 for each vertex j in the graph, do
 transMat[i, j] := transMat[i, j] OR (transMat[i, k]) AND transMat[k,j])
End-for
End-for
End-for
Display the transMat
End
```



(a)

$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

(b)

$$T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(c)

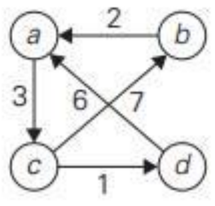
All pairs shortest path

floydWarshal(cost)

Input – The cost matrix of given Graph.

Output – Matrix to for shortest path between any vertex to any vertex.

```
Begin
 for k := 0 to n, do
 for i := 0 to n, do
for j := 0 to n, do
    if cost[i,k] + cost[k,j] < cost[i,j], then
              cost[i,j] := cost[i,k] + cost[k,j]
End-for
End-for
End-for
 display the current cost matrix End
```



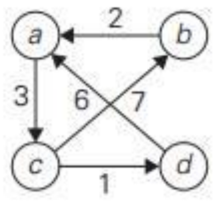
(a)

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

(b)

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(c)



(a)

$$W = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

(b)

$$D = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

(c)