6 : Graphs

IT 3206 – Data Structures and Algorithms

Level II - Semester 3

Overview

- This section introduces the graph data structure and its applications while comparing it with the tree structure.
- The section will also cover various types of graphs and graph representations.
- This section will further demonstrate two different algorithms for graph traversal, as well as multiple algorithms for finding the shortest paths, and their comparisons.

Intended Learning Outcomes

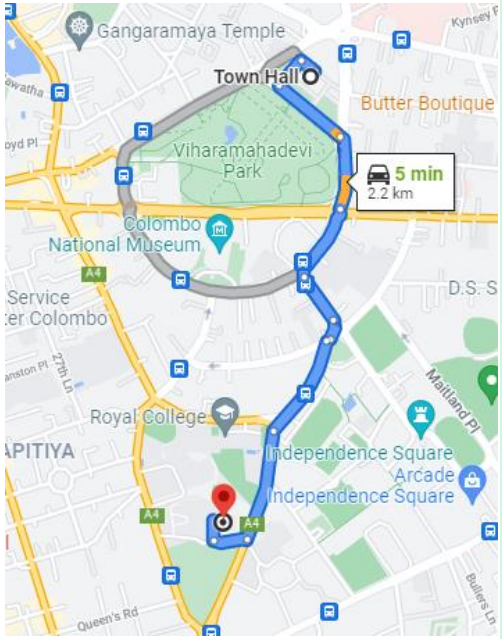
- At the end of this lesson, you will be able to;
  - Explain different types of graphs and their usage
  - Implement graphs and graph operations
  - Illustrate graph traversal and connectivity
  - Compare and implement different shortest path algorithms

List of Subtopics

- 6.1 Introduction to graphs
  - 6.1.1 Directed graphs, undirected graphs, weighted graphs
  - 6.1.2 Applications of graphs
- 6.2 Different types of Graph representations
  - 6.2.1 Array based implementation
  - 6.2.2 Linked-list based implementation
- 6.3 Graph traversal
  - 6.3.1 Depth first traversal (DFT)
  - 6.3.2 Breadth first traversal (BFT)
- 6.4 Shortest path algorithms
  - 6.4.1 Shortest path in unweighted graph
  - 6.4.2 Shortest path in weighted graph (Dijkstra's algorithm)
  - 6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm)

6.1 Introduction to graphs

- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph
- Graphs are a powerful and versatile data structure that easily allow you to represent real life relationships between different types of data (nodes).
- The best example of this would be google maps.



6.1 Introduction to graphs

- A graph G = (V,E) is a set of vertices (V) and a binary relation between vertices (set of edges – E).

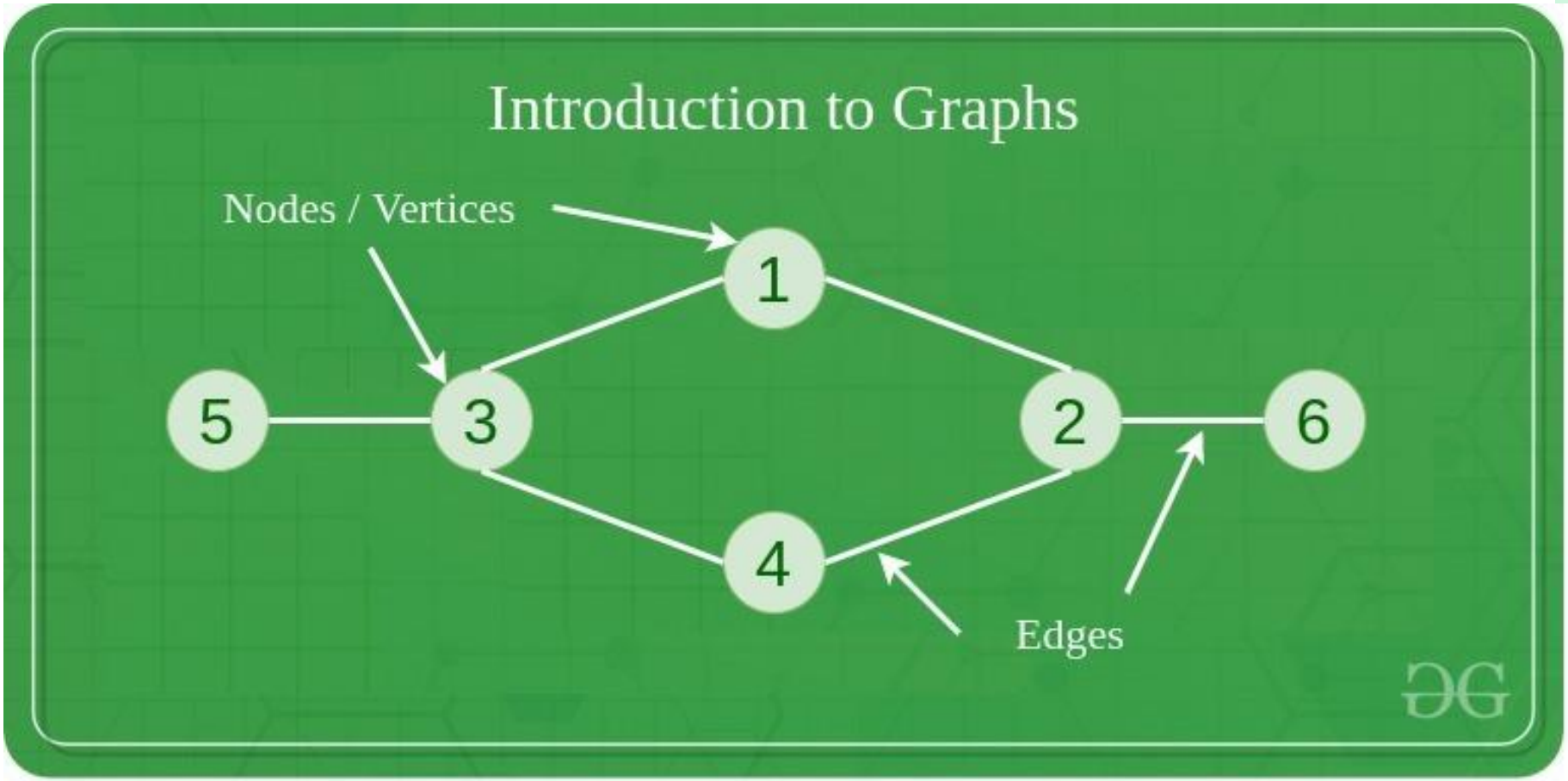
Components of a Graph

- **Vertices(Nodes):** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as nodes. Every node/vertex can be labelled or unlabeled.

- **Edges(Arcs):** Edges are the drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabeled.

6.1 Introduction to graphs contd.

Example



6.1 Introduction to graphs contd.

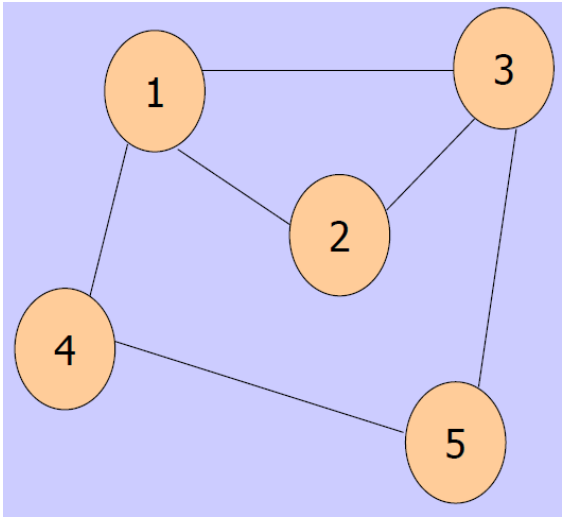
Some features of Trees vs Graphs

- Trees are connected and contains no cycles. Has V-1 number of edges.
- In the tree, there is exactly one root node, and every child can have only one parent. As against, in a graph, there is no concept of the root node.
- In a tree there exist only one path between any two vertices whereas a graph can have unidirectional and bidirectional paths between the nodes.
- A tree has a hierarchical structure whereas graph has a network model.
- A graph need not be a tree, but a tree must be a graph.

6.1 Introduction to graphs contd.

Graph Terminology

- **Adjacent vertices:** If (i,j) is an edge of the graph, then the nodes i and j are adjacent.
- An edge (i,j) is Incident to vertices i and j.

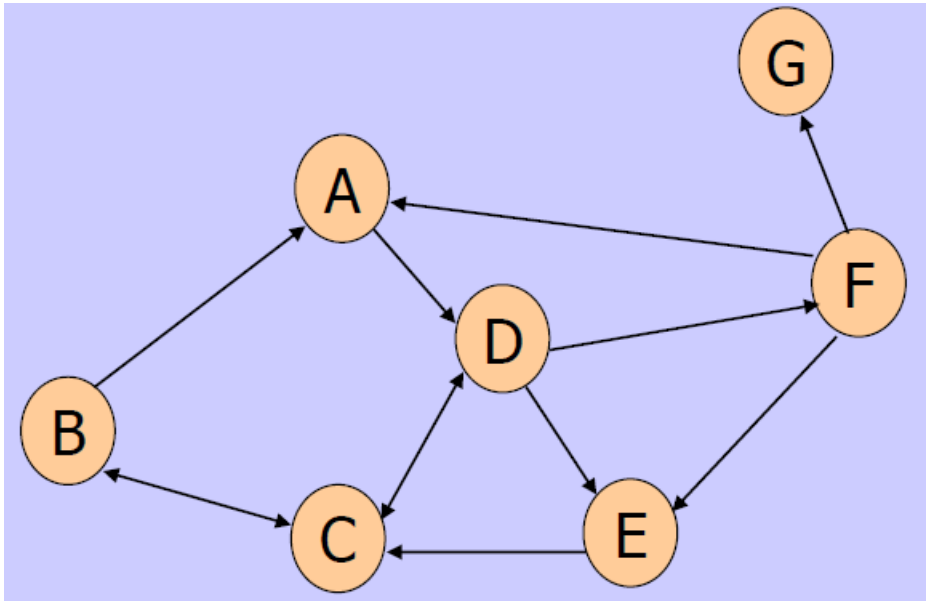


Vertices 2 and 5 are not adjacent

6.1 Introduction to graphs contd.

Graph Terminology contd.

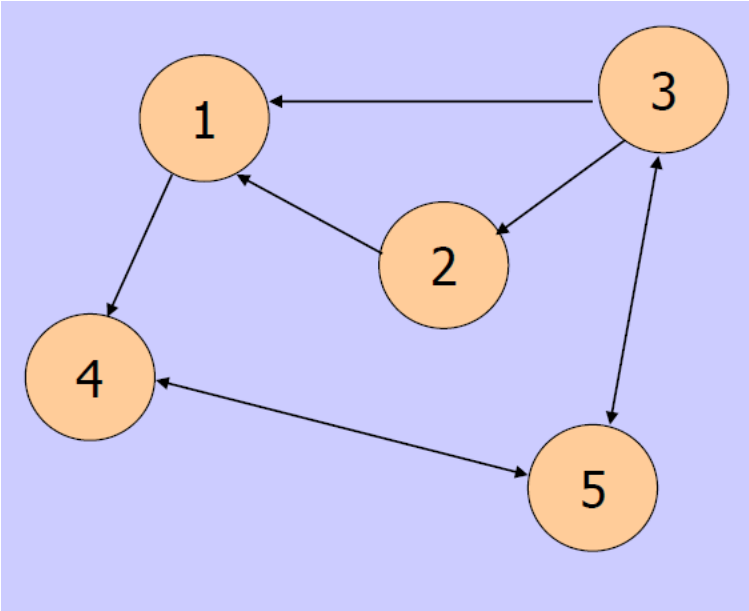
- **Path:** A sequence of edges in the graph
- There can be more than one path between two vertices
- Vertex A is reachable from B if there is a path from A to B



6.1 Introduction to graphs contd.

Graph Terminology contd.

- **Simple Path:** A path where all the vertices are distinct



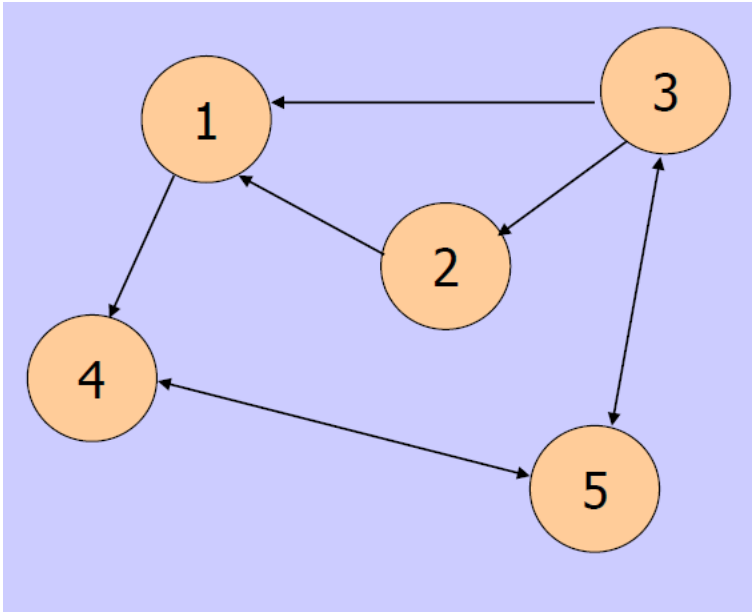
1,4,5,3 is a simple path.

But 1,4,5,4 is not a simple path.

6.1 Introduction to graphs contd.

Graph Terminology contd.

- **Length :** Sum of the lengths of the edges on the path.

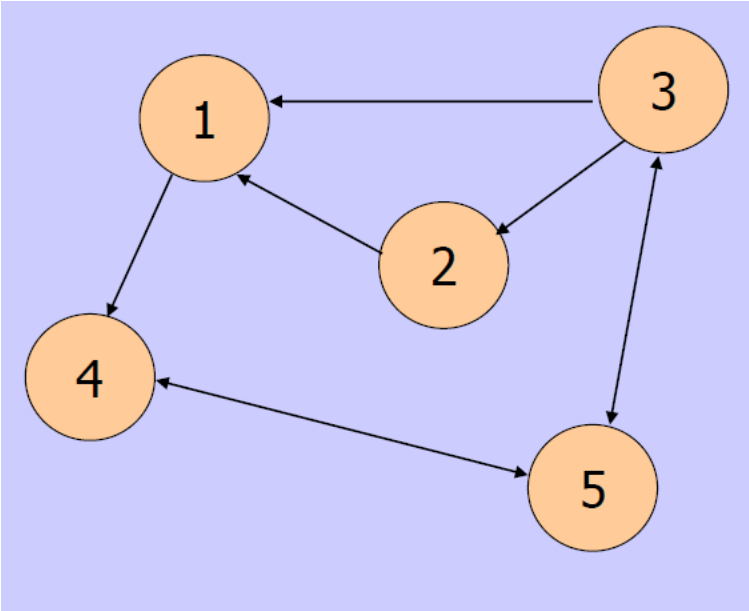


Length of the path 1,4,5,3 is 3

6.1 Introduction to graphs contd.

Graph Terminology contd.

- **Circuit:** A path whose first and last vertices are the same and no edge is repeated then the path is called a circuit.

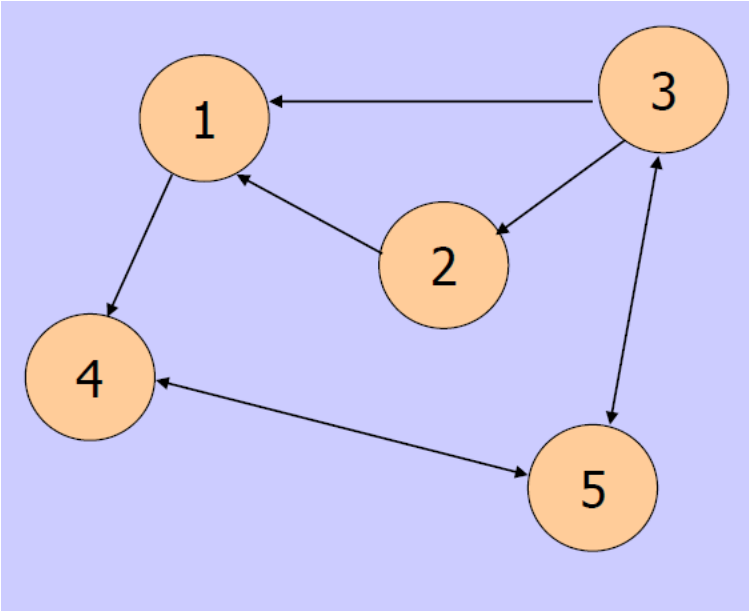


The path 3,2,1,4,5,3 is a circuit.

6.1 Introduction to graphs contd.

Graph Terminology contd.

- **Cycle:** A cycle where all the vertices are distinct except for the first (and the last) vertex.



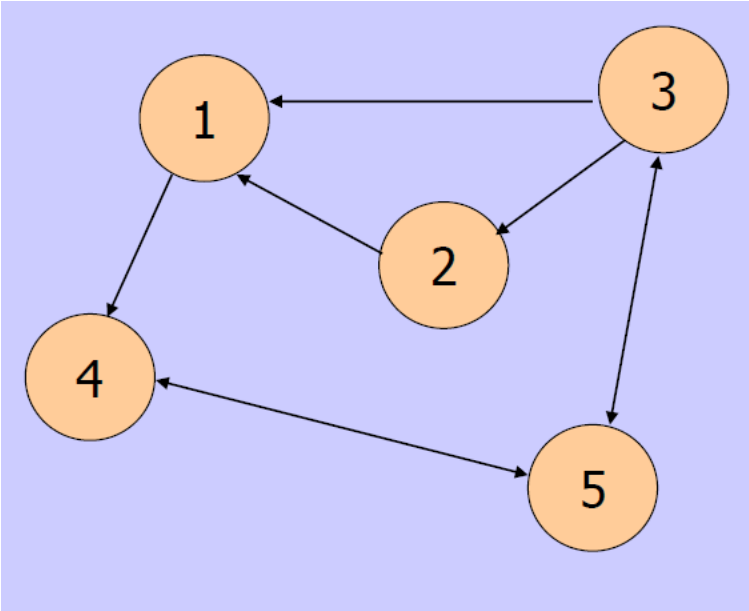
1,4,5,3,1 is a cycle

1,4,5,4,1 is not a cycle

6.1 Introduction to graphs contd.

Graph Terminology contd.

- **Hamiltonian Cycle:** A Cycle that contains all the vertices of the graph

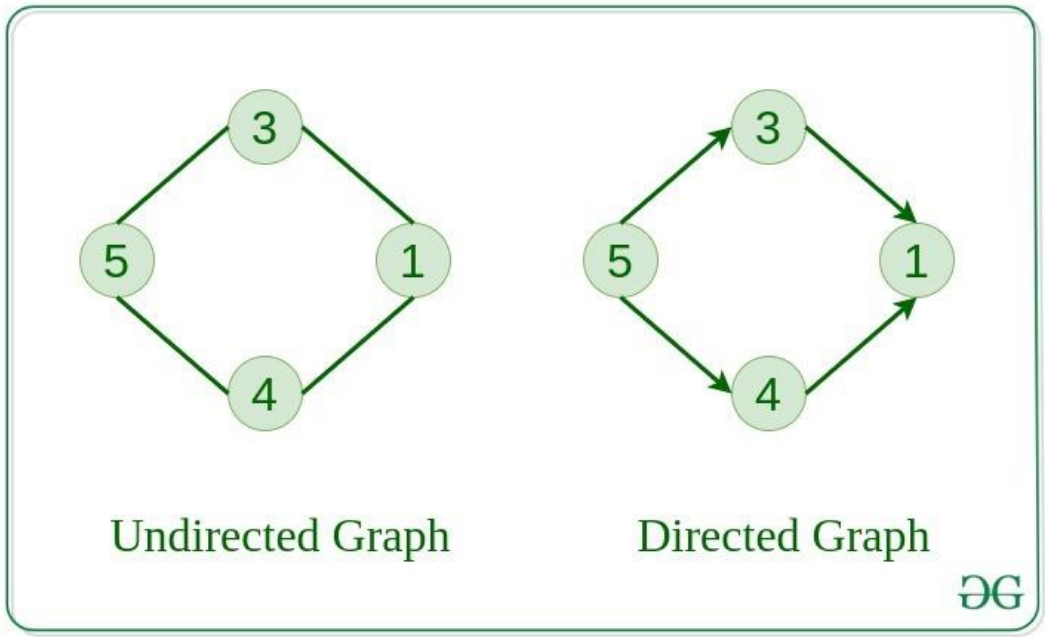


1,4,5,3,2,1 is a Hamiltonian Cycle

6.1.1 Directed graphs, undirected graphs, weighted graphs

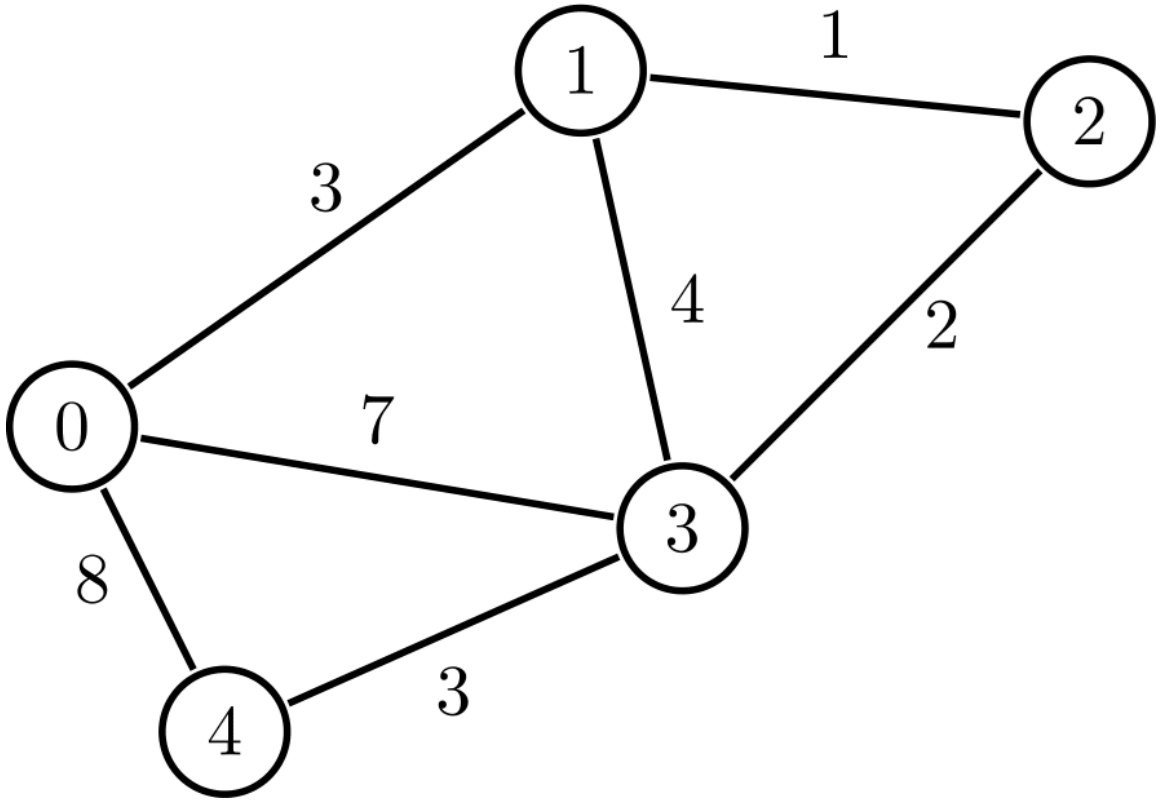
- **Directed Graph**
  - A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

- **Undirected Graph**
  - A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.



6.1.1 Directed graphs, undirected graphs, weighted graphs contd.

- **Weighted Graph**
  - A graph in which a weight, or number, associated with each edge.



6.1.1 Directed graphs, undirected graphs, weighted graphs contd.

- **Weighted Graph Example**
  - In this example, vertices represent cities, edges represent roads between cities and weight represents the distance of roads.

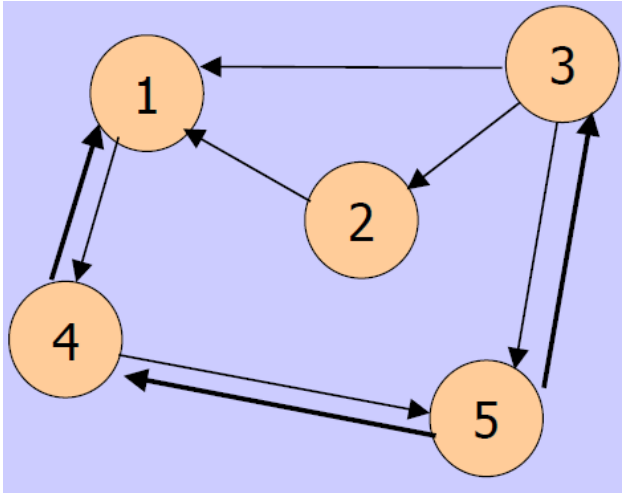
6.1.1 Directed graphs, undirected graphs, weighted graphs contd.

Graph Terminology

- **Degree of a Vertex** : In an undirected graph, the no. of edges incident to the vertex
- **In-degree:** The no. of edges entering the vertex in a directed graph
- **Out-Degree:** The no. of edges leaving the vertex in a directed graph

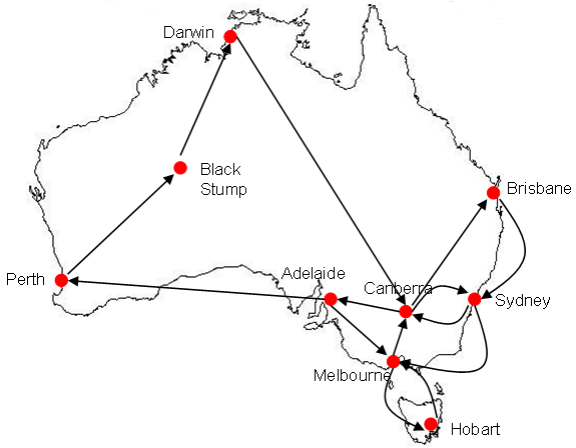
In-degree of 1 is 3

Out-degree of 1 is 1

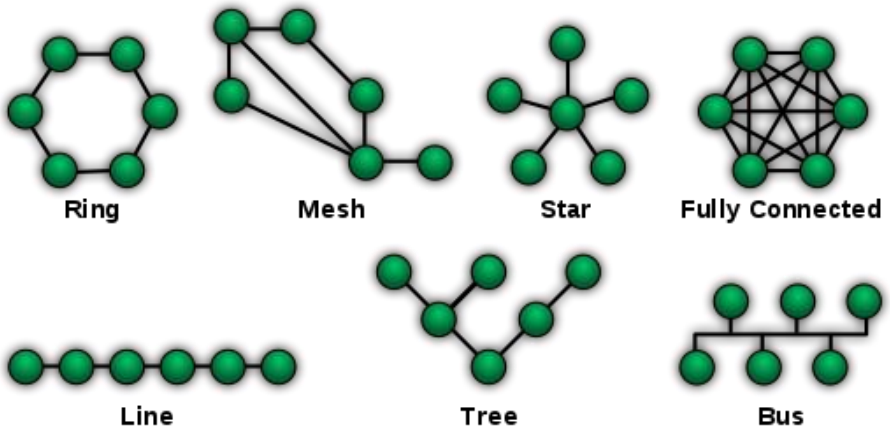


6.1.2 Applications of graphs

- Transportation networks: Highway network, Flight network



- Computer networks: Local area network, Internet, Web



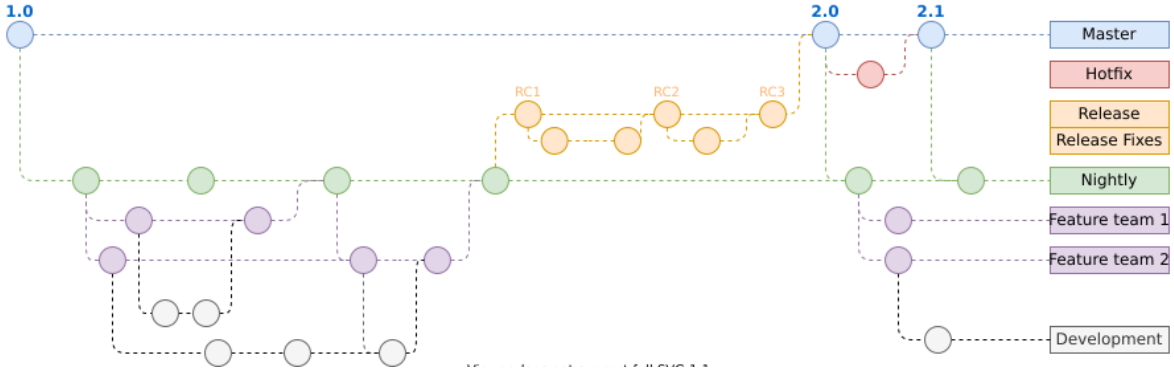
6.1.2 Applications of graphs contd.

- Social network graphs: Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures

.

6.1.2 Applications of graphs contd.

- Dependence graphs: Graphs can be used to represent dependencies or precedencies among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences



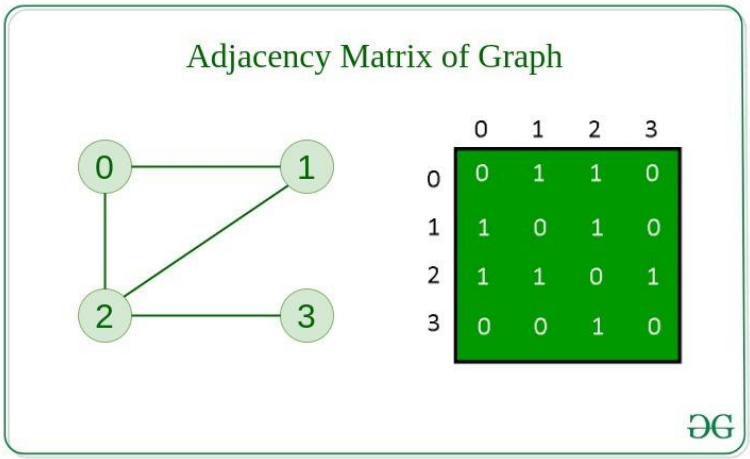
- Semantic networks. Vertices represent words or concepts, and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

6.2 Different types of Graph representations

- Generally, there are two types of graph representations.
  1. Array Based Implementation
  2. Linked List Based Implementation

6.2.1 Array based implementation

- **Adjacency Matrix**
  - In this method, the graph is stored in the form of the 2D matrix where rows and column denote vertices.
  - Each entry in the matrix represents the weight of the edge between those vertices.
  - The values of the Adjacency matrix are represented as follows: 1 or sometimes denoted as true. That indicates an edge from one node to another node; otherwise, 0 or false.

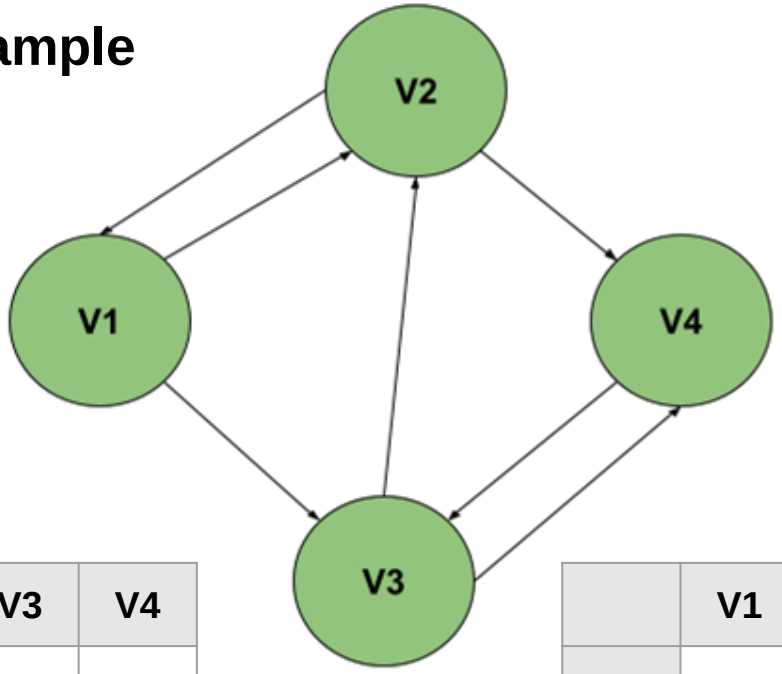


6.2.1 Array based implementation contd.

- **Path Matrix**
  - Path matrix has the same concept of adjacency matrix. But instead of having an edge from one node to another, this indicates whether there is a path from one node another.
  - The values of the path matrix are represented as follows: 1 or sometimes denoted as true. That indicates a path from one node to another node with the given length; otherwise, 0 or false.
  - For an example, path matrix of length 2 denotes whether there is a path from one node to another with length 2.
  - Hence, we can get the path matrix of length n by taking path matrix of length n-1 and multiplying it with path matrix of length 1.

6.2.1 Array based implementation contd.

- **Path Matrix Example**



	V1	V2	V3	V4
V1	0	1	1	0
V2	1	0	0	1
V3	0	1	0	1
V4	0	0	1	0

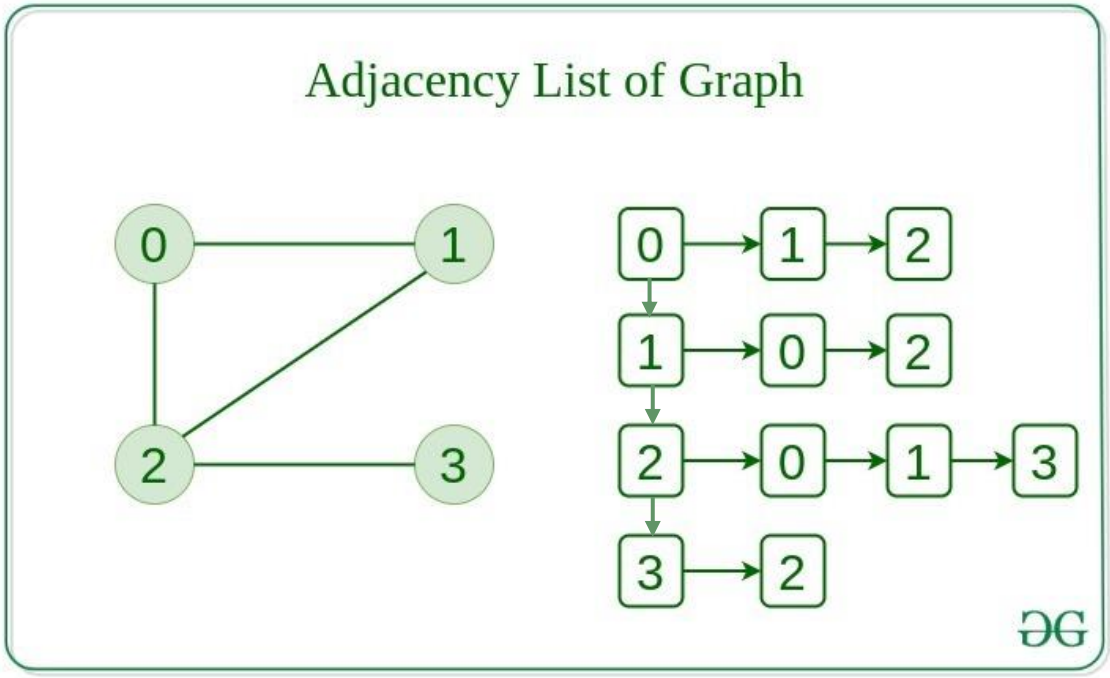
Path Matrix of Length 1

	V1	V2	V3	V4
V1	1	1	0	1
V2	0	1	1	0
V3	1	0	1	1
V4	0	1	0	1

Path Matrix of Length 2

6.2.2 Linked-list based implementation

- **Adjacency List**
  - This graph is represented as a collection of linked lists.
  - There is an array of pointer which points to the edges connected to that vertex.



6.3.1 Depth first traversal (DFT)

- Depth first traversal is a strategy used for exploring a graph.
  - Explore “deeper” in the graph whenever possible
  - Start from an arbitrary node.
  - Edges are explored out of the most recently discovered vertex v that still has unexplored edges.
  - When all of v’s edges have been explored, backtrack to the vertex from which v was discovered.
  - Stop when no unvisited nodes are found, and no backtracking can be done.
  - Implemented using a Stack.

6.3.1 Depth first traversal (DFT) contd.

- Used data structures.
  - color[v] : stores the color of the vertex (white: unvisited, grey: visited, black: processed)
  - parent[v] : stores the parent of v (if no parents null)
  - d[u] : stores discovered time
  - f[u] : stores finished time
  - time : stores current time

6.3.1 Depth first search (DFS) contd.

Algorithm

- Until there are no more undiscovered nodes.
  - Picks an undiscovered node and starts a depth first search from it.
  - The search proceeds from the *most recently discovered* node to discover new nodes.
  - When the last discovered node *v* is fully explored, backtracks to the node used to discover *v*. Eventually, the start node is fully explored.

6.3.1 Depth first search (DFS) Contd.

- In this version *all* nodes are discovered even if the graph is directed, or undirected and not connected
- The algorithm saves:
  - A depth first *forest* of the edges used to discover new nodes.
  - Timestamps for the first time a node *u* is discovered *d*[*u*] and the time when the node is fully explored *f*[*u*]

6.3.1 Depth first traversal (DFT) Contd.

Pseudocode Algorithm for DFT

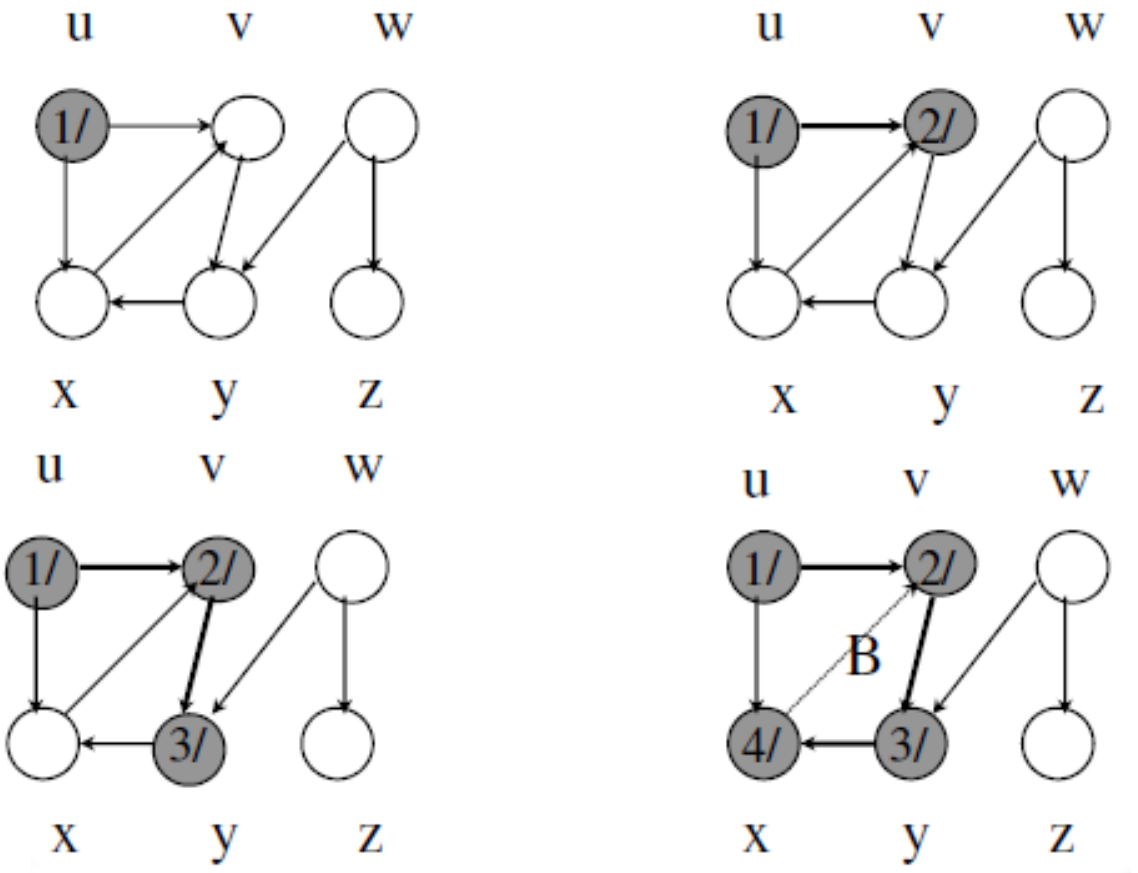
```
procedure DFT(G:graph; var color:carray; d, f:iarray;
        parent:parray);

        for each vertex u do
                color[u]:=white;
                parent[u]:=null;
        end for

        time:=0;

        for each vertex u do
                if color[u]=white then
                        DFT-Visit(u);
                end if
        end for

end DFT
```

6.3.1 Depth first traversal (DFT) Contd.

```
DFT-Visit(u)
        color[u]=:gray; time:=time+1; d[u]:=time
        for each v in adj[u] do
                if color[v]=white then
                        parent[v]:=u; DFT-Visit(v);
                end if
        end for
        color[u]:=black; time:=time+1; f[u]:=time;
end DFT-Visit
```

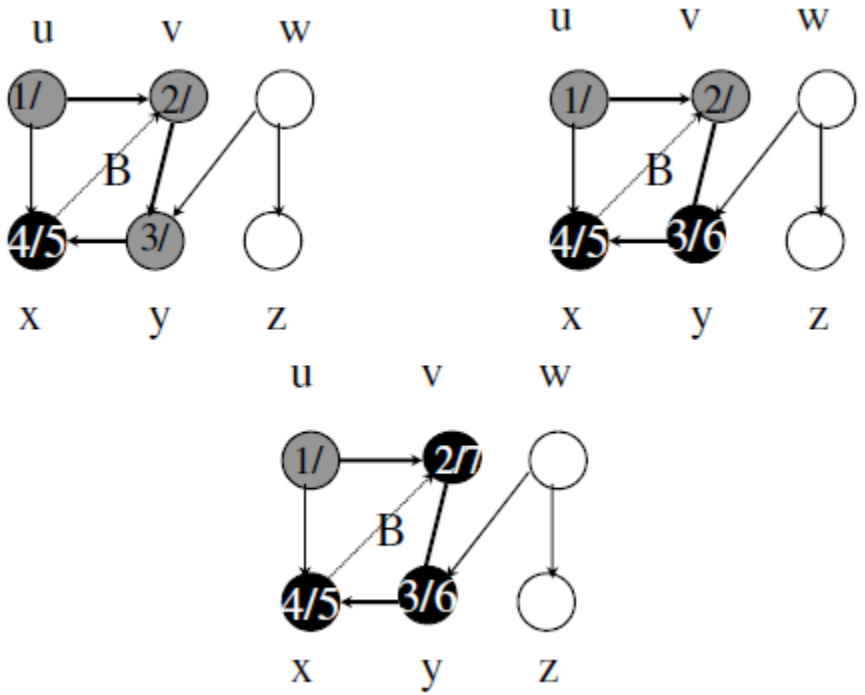
6.3.1 Depth first traversal (DFT) Contd.

Depth First Traversal Step by Step Example



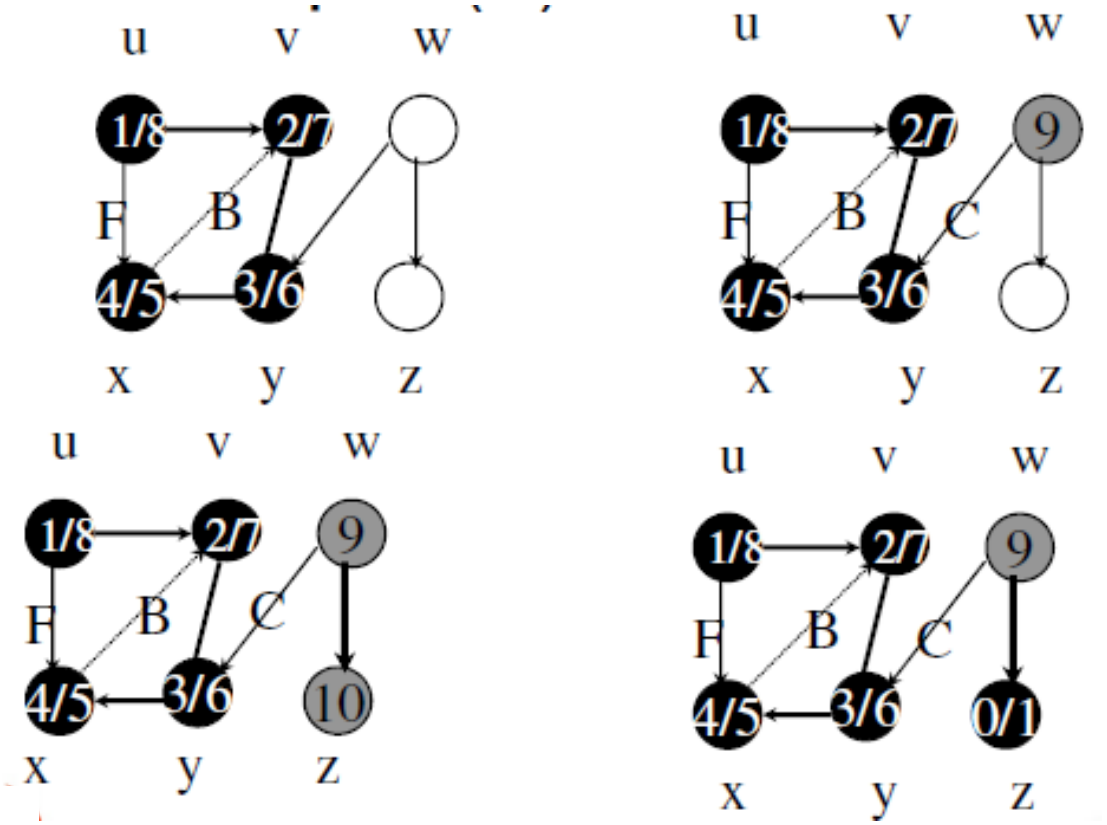
6.3.1 Depth first traversal (DFT) Contd.

Depth First Traversal Step by Step Example Contd.



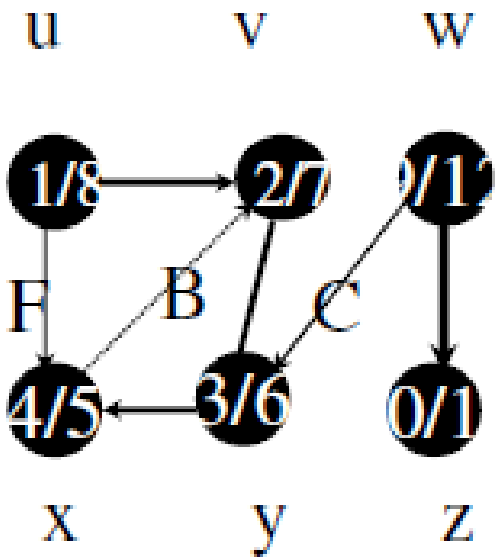
6.3.1 Depth first traversal (DFT) Contd.

Depth First Traversal Step by Step Example Contd.



6.3.1 Depth first traversal (DFT) Contd.

Depth First Traversal Step by Step Example Contd.

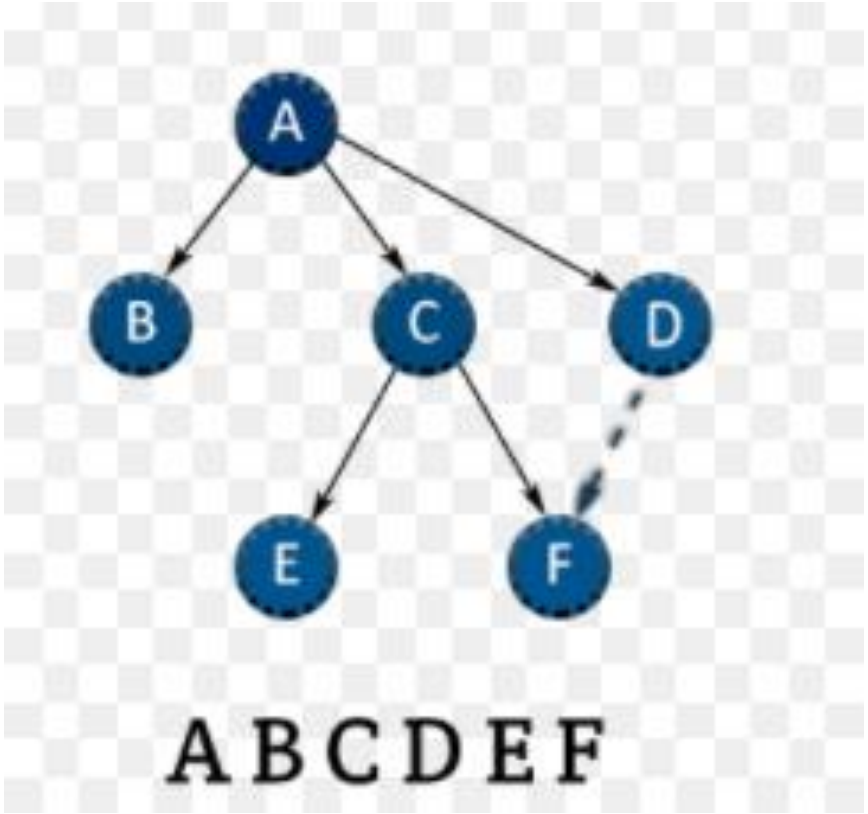


6.3.2 Breadth first traversal (BFT)

- Given a graph G=(V,E) and a source vertex s, BFT explores the edges of G to “discover” (visit) each node of G reachable from s.
- Idea - expand a frontier one step at a time.
- Frontier is a FIFO queue (O(1) time to update)
- Computes the shortest distance(dist) from s to any reachable node.
- Computes a breadth first tree (of parents) with root s that contains all the reachable vertices from s.
- To get O(|V|+|E|) we use an adjacency list representation. If we used an adjacency matrix it would be $O(|V|^2)$

6.3.2 Breadth first traversal (BFT) contd.

- Example for Breadth First Traversal



6.3.2 Breadth first traversal (BFT) contd

Coloring the nodes

- We use colors (white, gray and black) to denote the state of the node during the search.
- A node is white if it has not been reached (discovered).
- Discovered nodes are gray or black. Gray nodes are at the frontier of the search.
- Black nodes are fully explored nodes.

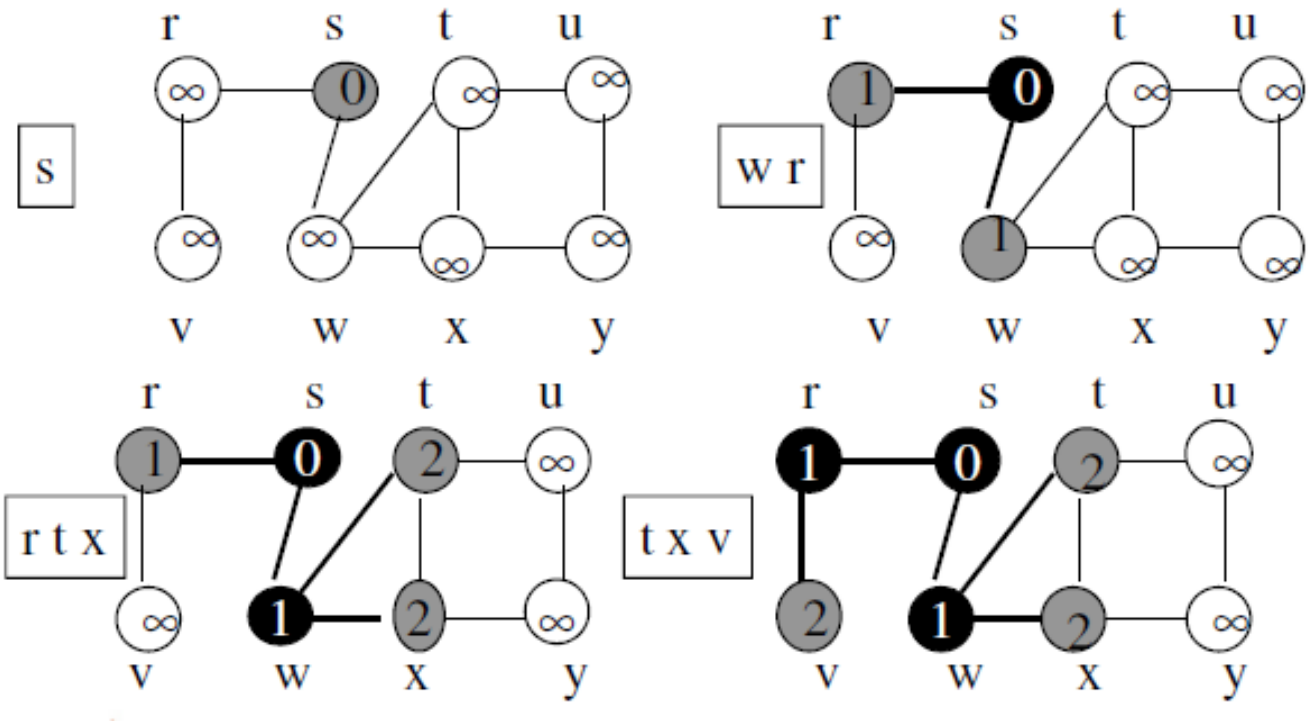
6.3.1 Breath first traversal (BFT) Contd.

Pseudocode Algorithm for BFT

```
procedure BFT(G:graph; s:node; var color:carray;
                dist:iarray; parent:parray);

        for each vertex u do
            color[u]:=white; dist[u]:=∞;
            parent[u]:=null;
        end for
        color[s]:=gray; dist[s]:=0;
        init(Q); enqueue(Q, s);
        while not (empty(Q)) do
            u:=head(Q);
            for each v in adj[u] do
                if color[v]=white then O(E)
                        color[v]:=gray; dist[v]:=dist[u]+1;
                        parent[v]:=u; enqueue(Q, v);
                end if
            end for
            dequeue(Q); color[u]:=black;

        end while
end BFT
```

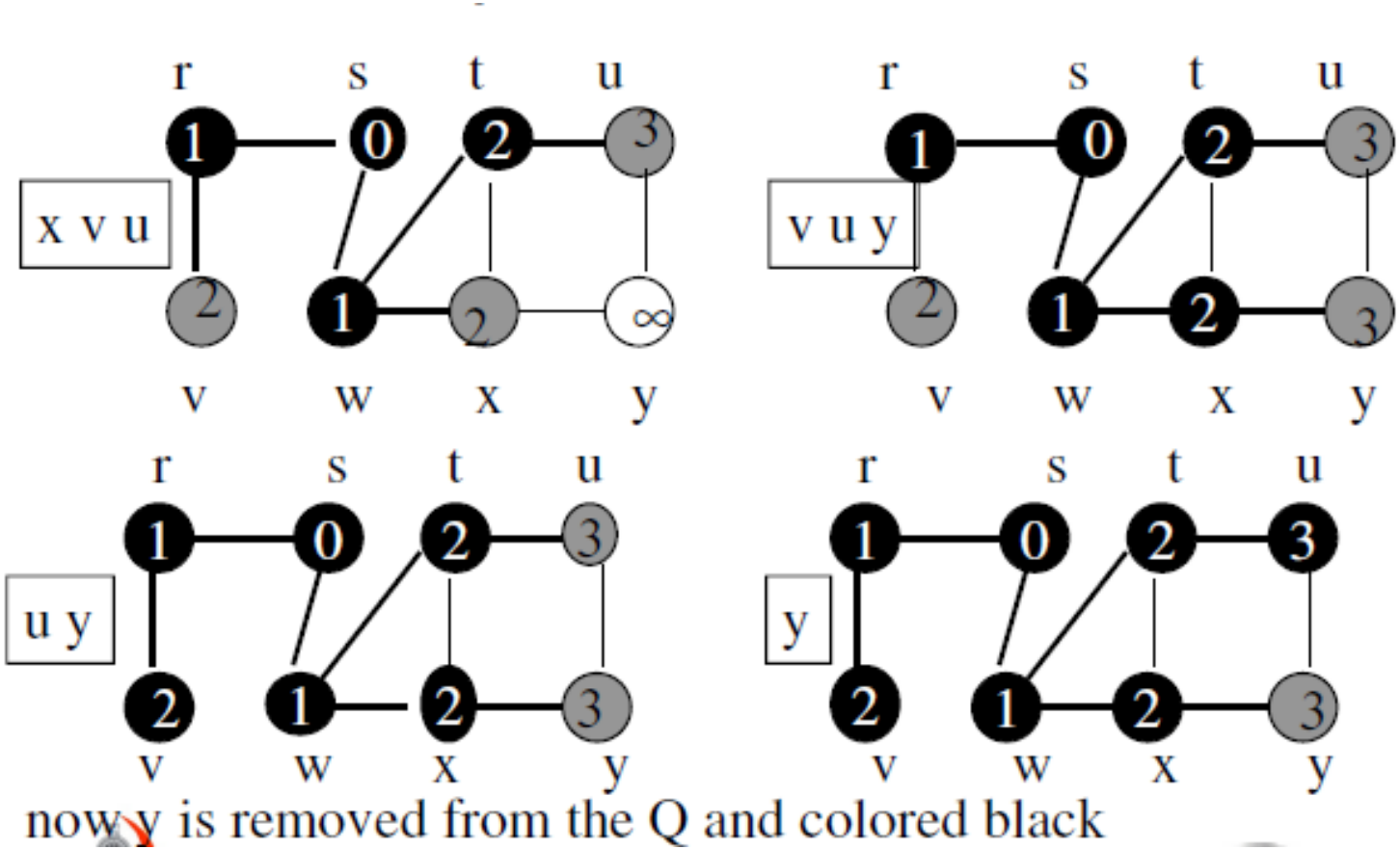
6.3.2 Breadth first search (BFS) contd.

Breadth First Search Step by Step Example



6.3.2 Breadth first search (BFS) contd.

B



now y is removed from the Q and colored black

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm)

- The algorithm characterizes each node by its state
- The state of a node consists of two features:
  - **distance value** and **status label**
- Distance value of a node is a scalar representing an estimate of the its distance from node s.
- Status label is an attribute specifying whether the distance value of a node is equal to the shortest distance to node s or not.
  - The status label of a node is **Permanent** if its distance value is equal to the shortest distance from node s
  - Otherwise, the status label of a node is **Temporary**
- The algorithm maintains and step-by-step updates the states of the nodes
- At each step one node is designated as **current**

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Notation

In what follows:

- $\mathbf{d_l}$ denotes the distance value of a node **l**.
- **p** or **t** denotes the status label of a node, where **p** stand for permanent, and **t** stands for temporary
- $\mathbf{c_{ij}}$ is the cost of traversing link **(i, j)** as given by the problem

The state of a node **l** is the ordered pair of its distance value $\mathbf{d_l}$ and its status label.

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Algorithm

- **Step 1** : Initialization
- **Step 2** : Distance Value Update and Current Node Designation Update
- **Step 3** : Termination Criterion

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Step 1 : Initialization

- Assign the zero-distance value to node s, and label it as **Permanent**. [The state of node s is (0, p)]
- Assign to every node a distance value of ∞ and label them as **Temporary**. [The state of every other node is (∞, t)]
- Designate the node s as the **current** node

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Step 2 : Distance Value Update and Current Node Designation Update

Let i be the index of the current node.

- Find the set J of nodes with temporary labels that can be reached from the current node i by a link (i, j). Update the distance values of these nodes.
- Determine a node j that has the smallest distance value dj among all nodes $j \in J$.
- Change the label of node j to permanent and designate this node as the current node.

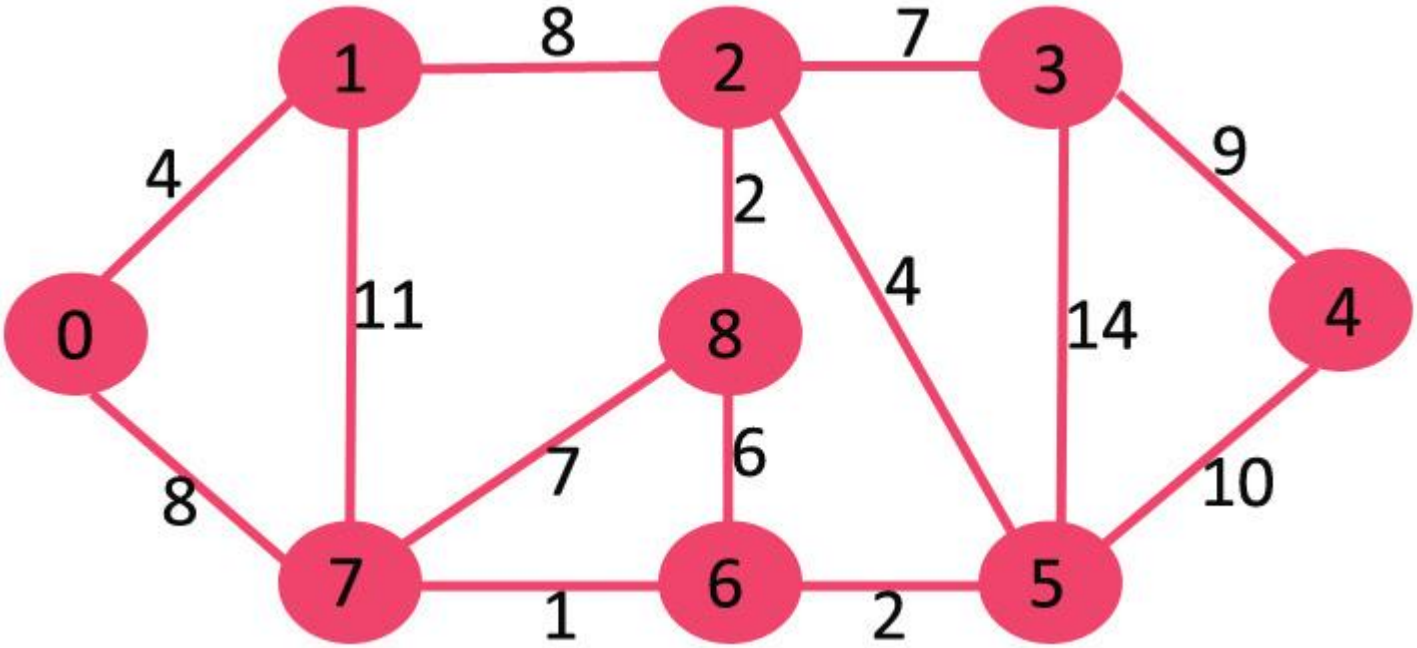
6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Step 3 : Termination Criterion

- If all nodes that can be reached from node s have been permanently labeled, then stop.
- If algorithm cannot reach any temporary labeled node from the current node, then all the temporary labels become permanent, then stop
- Otherwise, go to step 2.

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example



6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.

- The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.

- Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices.

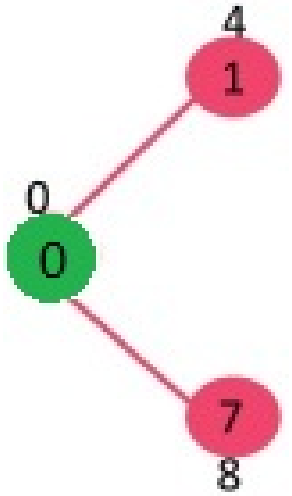
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

- Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.

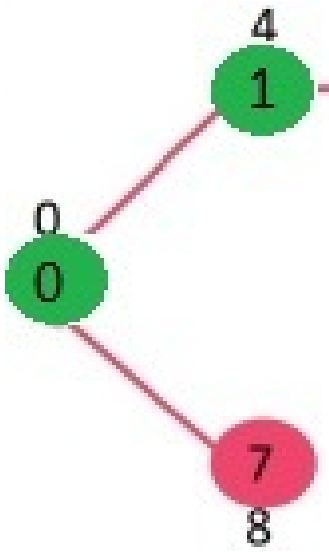
0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞	∞
	4						8	



- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.



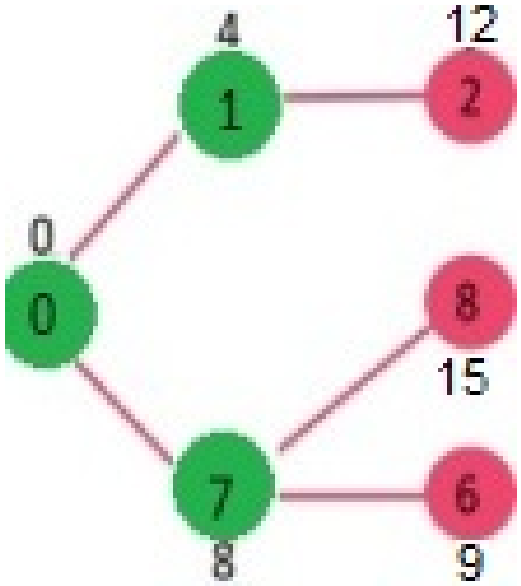
0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞	∞
	4						8	
		12						

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.

0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞	∞
	4	∞	∞	∞	∞	∞	**8**	∞
		12	∞	∞	∞	∞	**8**	∞
		12	∞	∞	∞	**9**	∞	**15**

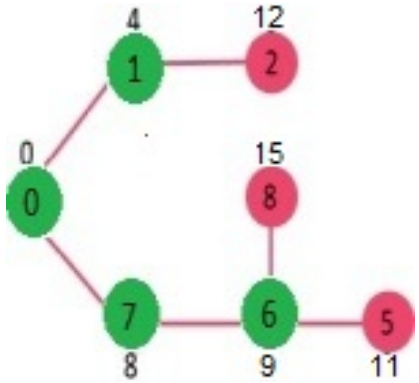


- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.

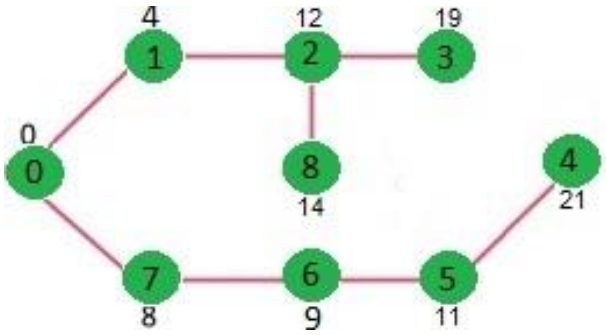
0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞	∞
	4	∞	∞	∞	∞	∞	**8**	∞
		12	∞	∞	∞	∞	**8**	∞
		12	∞	∞	∞	**9**	∞	**15**
		12	∞	∞	**11**		∞	**15**



- We repeat the above steps until sptSet does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

6.4.2 Shortest path in weighted graph (Dijkstra's algorithm) contd.

Example contd.



0	1	2	3	4	5	6	7	8
0	∞	∞	∞	∞	∞	∞	∞	∞
	4	∞	∞	∞	∞	∞	8	∞
		12	∞	∞	∞	∞	8	∞
		12	∞	∞	∞	9		15
		12	∞	∞	11			15
		12	25	21				15
			19	21				14
			19	21				
				21				

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm)

Algorithm

- **Input:** Graph and a source vertex src

- **Output:** Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

  Negative weight cycle is a cycle with weights that sum to a negative number. If there is a negative weight cycle, algorithm can go on relaxing its nodes indefinitely.

- **Step 01:** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Algorithm contd.

- **Step 02:** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
    - Do following for each edge u-v
        - If dist[v] > dist[u] + weight of edge uv, then update dist[v]
            - dist[v] = dist[u] + weight of edge uv

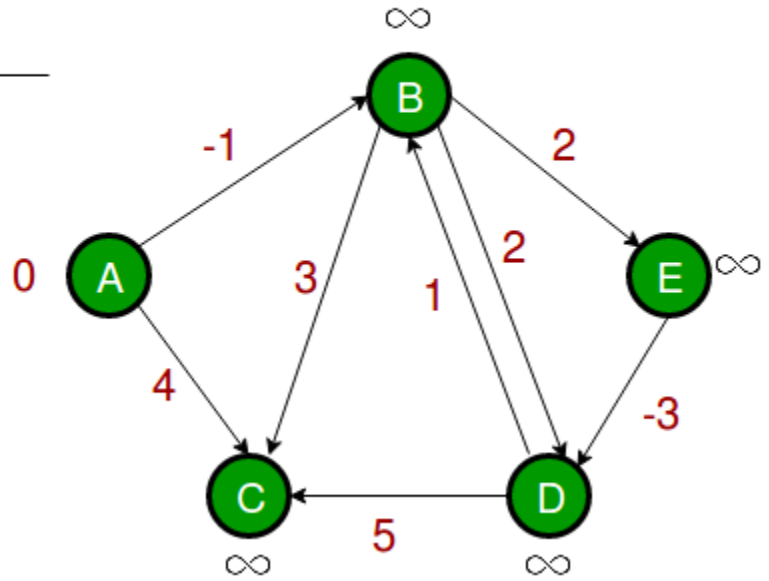
- **Step 03:** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
    - If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Example

- Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

A	B	C	D	E
0	∞	∞	∞	∞



6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Example contd.

- Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).

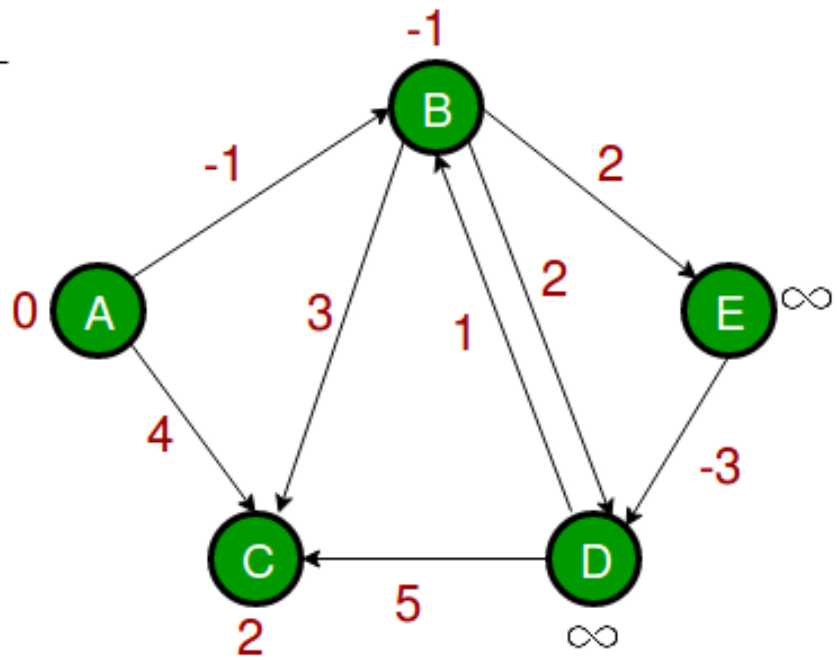
- We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed.

- The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Example contd.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

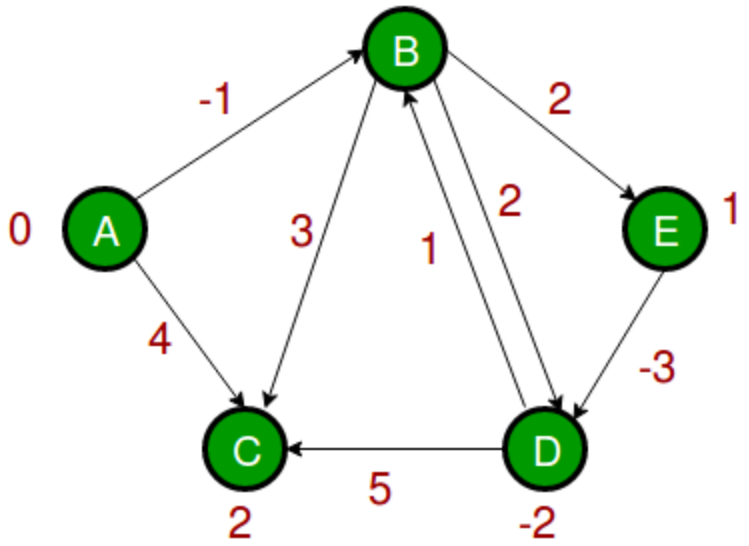


6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Example contd.

- The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

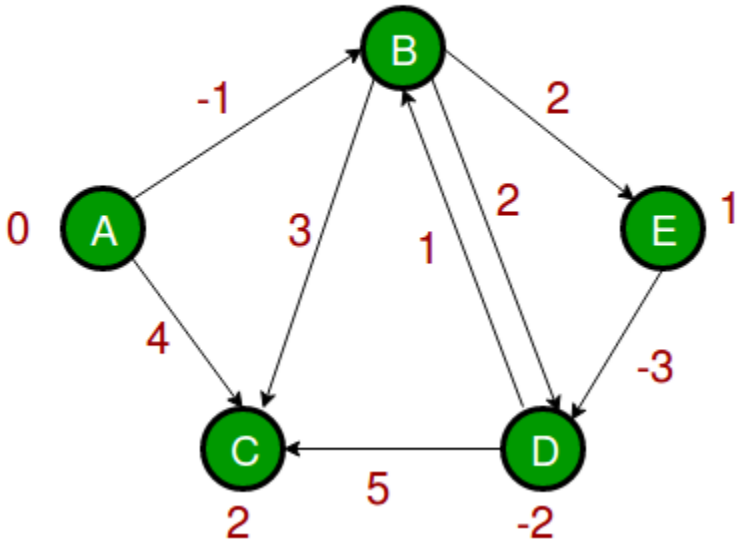


6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

Example contd.

- The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

```
void BellmanFord(Graph graph, int src) {
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];

    // Step 01
    // Step 02
    // Step 03

}
```

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

```
// Step 1: Initialize distances from src to all other
// vertices as INFINITE
for (int i = 0; i < V; ++i)
    dist[i] = Integer.MAX_VALUE;
dist[src] = 0;
```

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

```
// Step 2: Relax all edges |V| - 1 times. A simple
// shortest path from src to any other vertex can
// have at-most |V| - 1 edges
for (int i = 1; i < V; ++i) {
    for (int j = 0; j < E; ++j) {
        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}
```

6.4.3 Shortest path weighted graph with negative edges (Bellman-Ford Algorithm) contd.

```
// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
// contain negative weight cycle. If we get a shorter
// path, then there is a cycle.
for (int j = 0; j < E; ++j) {
    int u = graph.edge[j].src;
    int v = graph.edge[j].dest;
    int weight = graph.edge[j].weight;
    if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v]) {
        System.out.println("Graph contains negative weight cycle");
        return;
    }
}
```

Summary

Introduction to Graphs	Directed Graphs, Undirected Graphs and Weighted Graphs
Different Types of Graph Representations	Adjacency Matrix, Path Matrix and Adjacency list.
Graph Traversal	Depth First Traversal and Breath First Traversal
Shortest Path Algorithms in Graphs	Dijkstra's Algorithm and Bellman-Ford Algorithm