DATA STRUCTURES (ITPC-203)

Graphs – Part I

Dr. Sanga Chaki

Department of Information Technology

Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

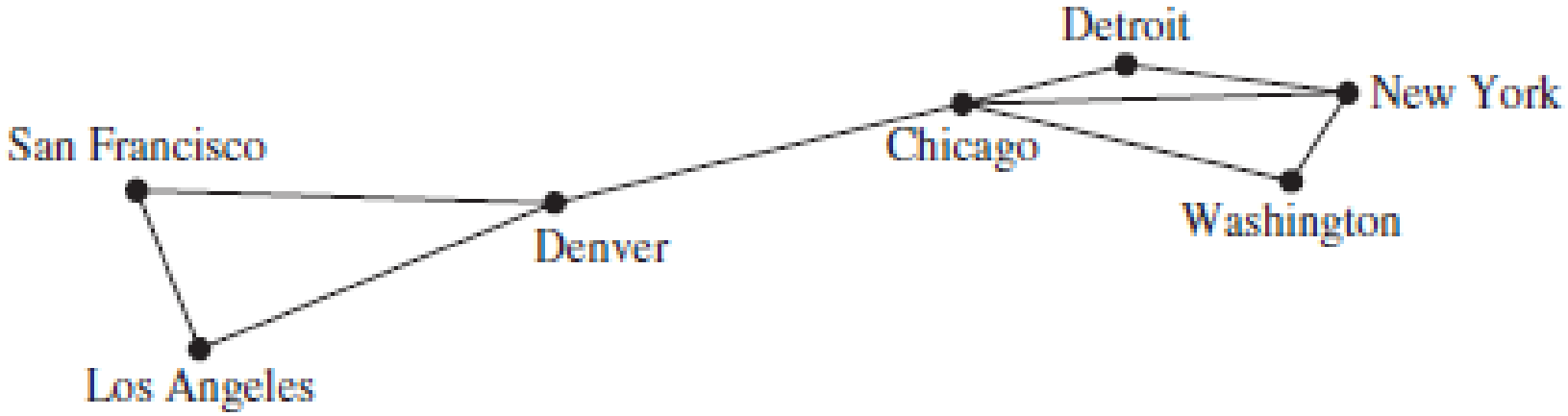
- What are graphs
- Applications
- Basic Terminology
- Finite and infinite graph
- Directed and undirected graphs
- Representations of Graphs: Adjacency Matrices, Adjacency List
- Graph Traversal : Breadth First Search

Graphs

1. Graphs are non-linear data structures consisting of vertices and edges that connect these vertices.
2. Where used? everywhere:
   - to model social media connections
   - Model telephone calls between telephone numbers,
   - model links between websites.
   - model roadmaps – Google Map
   - finding the shortest path between two cities in a transportation network.
   - Schedule exams
   - to represent who influences whom in an organization,

Graph Terminology

1. Formally, A graph G = (V ,E) consists of
   - V , a nonempty set of vertices (or nodes) and
   - E, a set of edges.
2. Each edge has either one or two vertices associated with it, called its endpoints.
3. An edge is said to connect the vertices/nodes that are its endpoints.

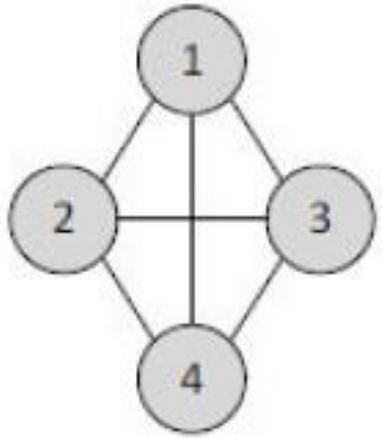


Types: Finite and Infinite Graph

1. The set of vertices V of a graph G may be infinite.
2. A graph with an infinite vertex set or an infinite number of edges is called an infinite graph.
3. A graph with a finite vertex set and a finite edge set is called a finite graph

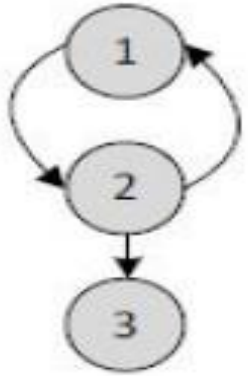
Types: Directed and Undirected Graph

1. Undirected Graph:
   - In an undirected graph the pair of vertices representing any edge is unordered.
   - Thus, the pairs (v1, v2) and (v2, v1) represent the same edge.
2. Directed Graph/Digraph:
   - In a directed graph each edge is represented by a directed pair of vertices.
   - If the edge is <v1, v2>, then v1 is the tail and v2 the head of the edge.
   - This is not the same as <v2, v1>
   - The edges of a directed graph are drawn with an arrow from the tail to the head.



G1 – Undirected Graph

Set of vertices = { 1, 2, 3, 4 }
Set of edges = { (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4) }

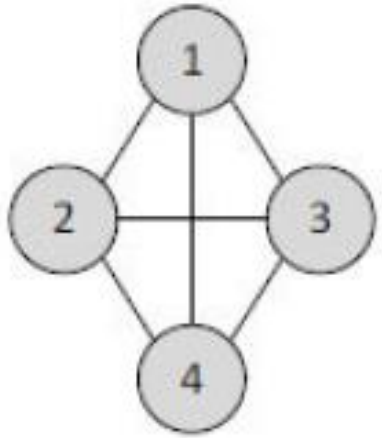


G2 – Directed Graph

Set of vertices = { 1, 2, 3 }
Set of edges = { <1, 2>, <2, 1>, <2, 3> }

Terminology for Undirected Graph:

1. Adjacent Vertices:
   - In an undirected graph if (v1, v2) is an edge in the set of edges, then the vertices v1 and v2 are said to be adjacent
2. Incident Edges:
   - And the edge (v1, v2) is said to be incident on vertices v1 and v2
3. Example: v1 is adjacent to v2, v3 and v4.
4. The edges incident on v4 are (v1,v4), (v2,v4) and (v3,v4)

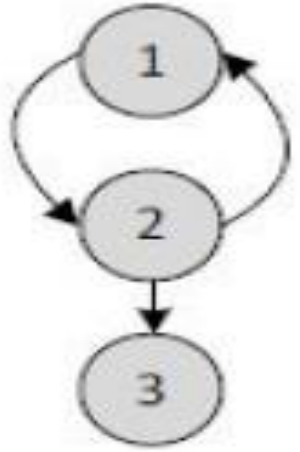


G1 – Undirected Graph

Set of vertices = { 1, 2, 3, 4 }
Set of edges = { (1, 2), (1, 3), (1, 4),
(2, 3), (2, 4), (3, 4) }

Terminology for Digraph

1. When (u, v) is an edge of a digraph, u is said to be adjacent to v and v is said to be adjacent from u.
2. Edge (u, v) is incident to v from u.
3. Vertex u is called the initial vertex of (u, v), and v is called the terminal/end vertex of (u, v).
4. Example:
   - v1 is adjacent to v2.
   - v2 is adjacent to v1.
   - V2 is also adjacent to v3.



G2 – Directed Graph

Set of vertices = { 1, 2, 3 }
Set of edges = { <1, 2>, <2, 1>, <2, 3> }

Graph Representations

1. How to represent a graph in memory?
2. Which is better?
   - One of these representations will be better than others for a given application.
3. The most commonly used representations for graphs are

   (a) Adjacency matrix

   (b) Adjacency lists

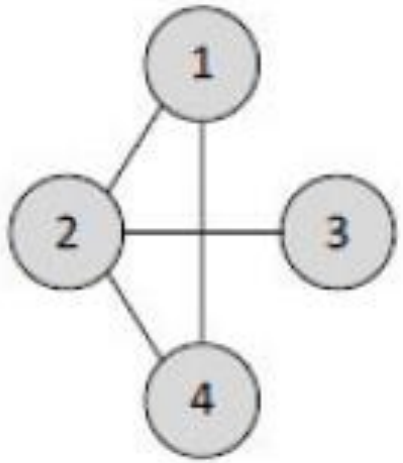
   (c) Adjacency multi-lists – later

Adjacency Matrices

1. To simplify computation, graphs can be represented using matrices.
2. Suppose that G = (V ,E) is a simple graph where |V| = n.
3. Suppose that the vertices of G are listed as $v_1, v_2, \ldots, v_n$.
4. The adjacency matrix A (or A_G) of G, with respect to this listing of the vertices, is the **n x n zero–one matrix** A = [a_{ij}], where

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

Adjacency Matrices - Example



	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

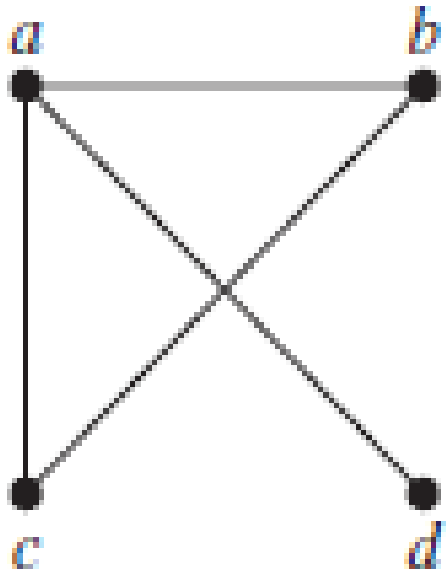
(a) Adjacency matrix for undirected graph

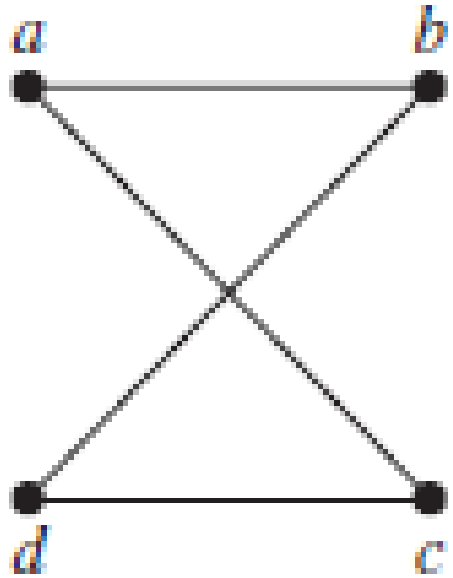
	1	2	3
1	0	1	0
2	1	0	1
3	0	0	1

(b) Adjacency matrix for directed graph

Adjacency Matrices

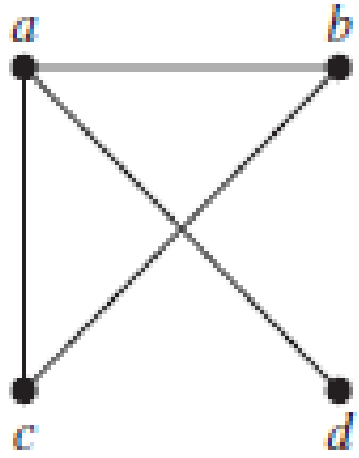
1. Use an adjacency matrix to represent the graphs

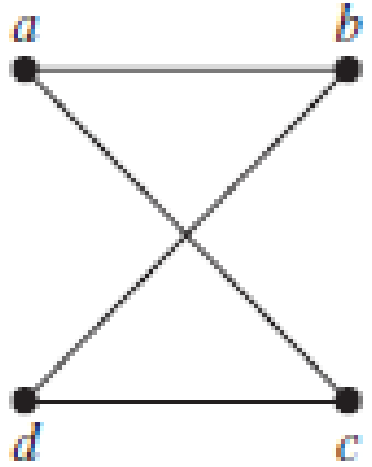




Adjacency Matrices

1. Use an adjacency matrix to represent the graphs





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

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

Adjacency Matrices - Properties

1. The adjacency matrix for an undirected graph is symmetric.
2. The adjacency matrix for a directed graph need not be symmetric.
3. The space needed to represent a graph using its adjacency matrix is the space for nXn locations.
4. About half of this space can be saved in the case of undirected graphs by storing only the upper or lower triangle elements of the matrix.

Adjacency lists

1. An adjacency list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.
2. Vertex-based representation
3. Make a list of all the vertices in a graph
4. Specify the vertices that are adjacent to each vertex of the graph.
5. Exar

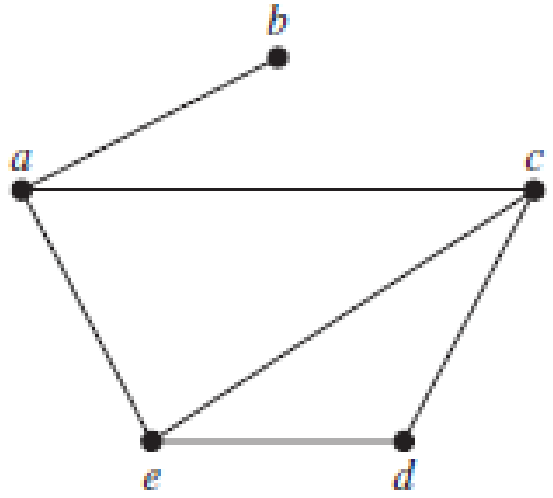


TABLE 1 An Adjacency List for a Simple Graph.

Vertex	*Adjacent Vertices*
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

Adjacency lists

1. Example: Represent the directed graph shown in the figure by listing all the vertices that are the terminal vertices of edges starting at each vertex of the graph.

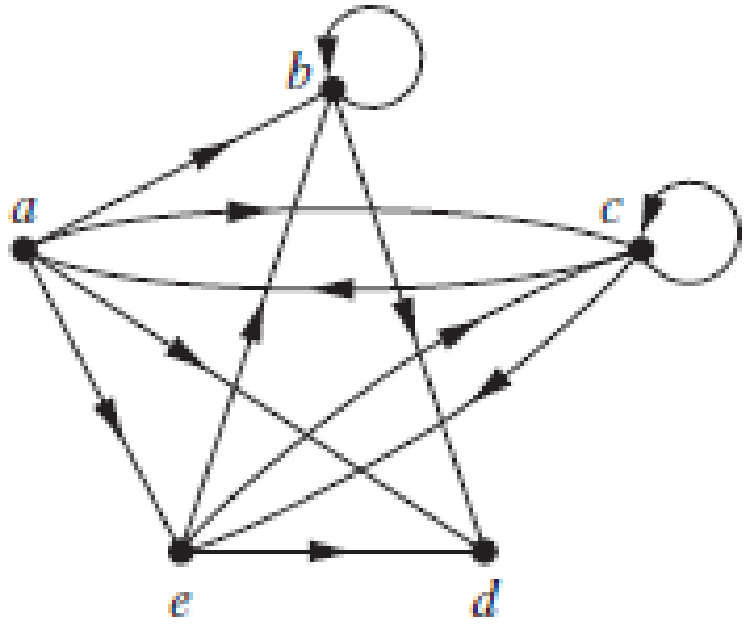


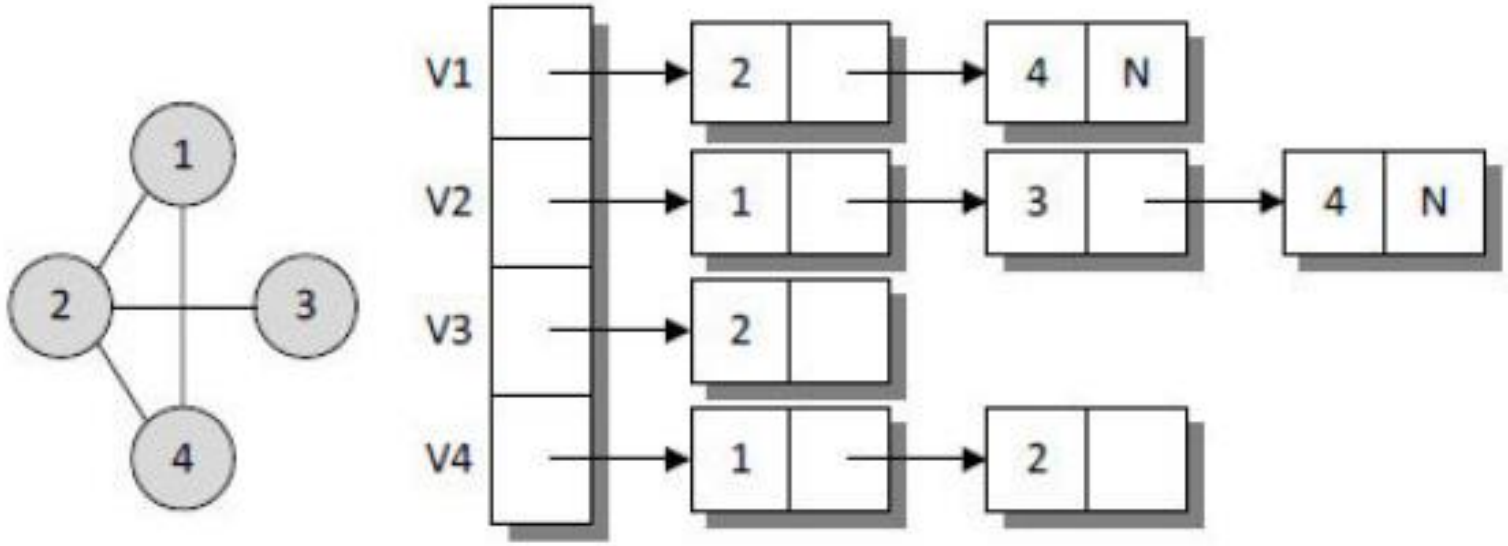
TABLE 2 An Adjacency List for a Directed Graph.

Initial Vertex	*Terminal Vertices*
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

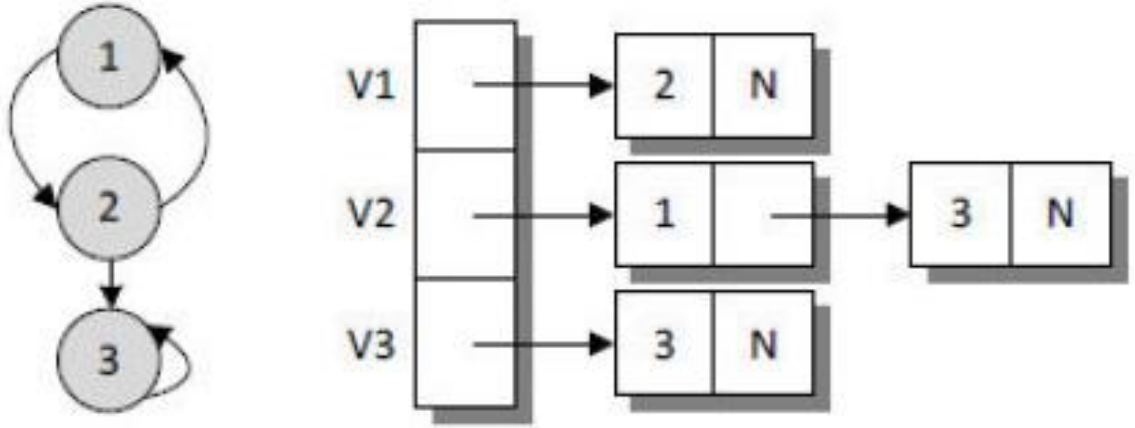
Adjacency Lists - update

1. To represent graphs using adjacency lists in memory, associate each vertex with a linked list of vertices adjacent to it.
2. Normally an array is used to store the vertices.
3. Each array element contains
   a) the vertex label,
   b) any other related information,
   c) plus a pointer to a linked list of nodes containing adjacent vertices.
4. The array provides random access to the adjacency list for any particular vertex.

Adjacency Lists

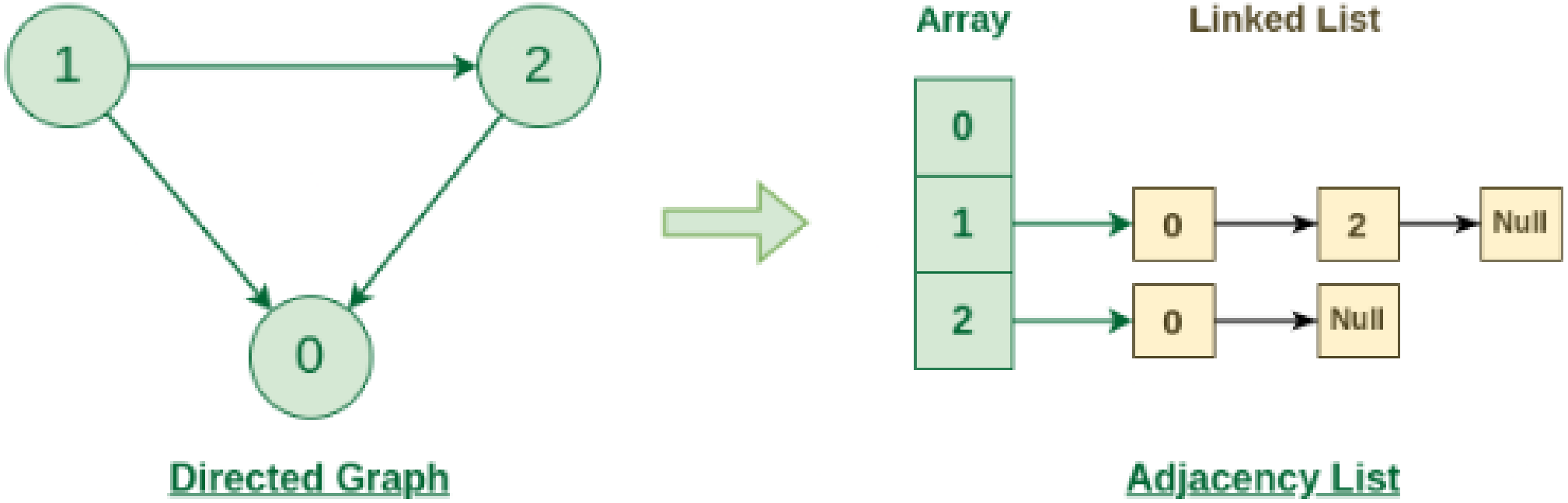


(a) Adjacency lists for undirected graph



(b) Adjacency lists for directed graph

Adjacency Lists



Adjacency Lists - Properties

1. The advantage of this representation is that
   a) Possible to quickly find all the edges associated with a given vertex by traversing the list
   b) Do not have to look through possibly hundreds of zero values to find a few ones in a row of an adjacency matrix.
   c) The number of graph edges is easily computed.
   d) The adjacency list is a jagged array.
2. Disadvantage:
   a) Might not be suitable for graphs with many edges. Adjacency matrices might be better in those cases.
   b) Any more?

Graph Traversals

1. Given a vertex in a directed or undirected graph we may wish to visit all vertices in the graph that are reachable from this vertex.
2. This can be done in two ways:
   a) Depth First Search algorithm
   b) Breadth First Search algorithm.

Breadth First Search (BFS) Algorithm

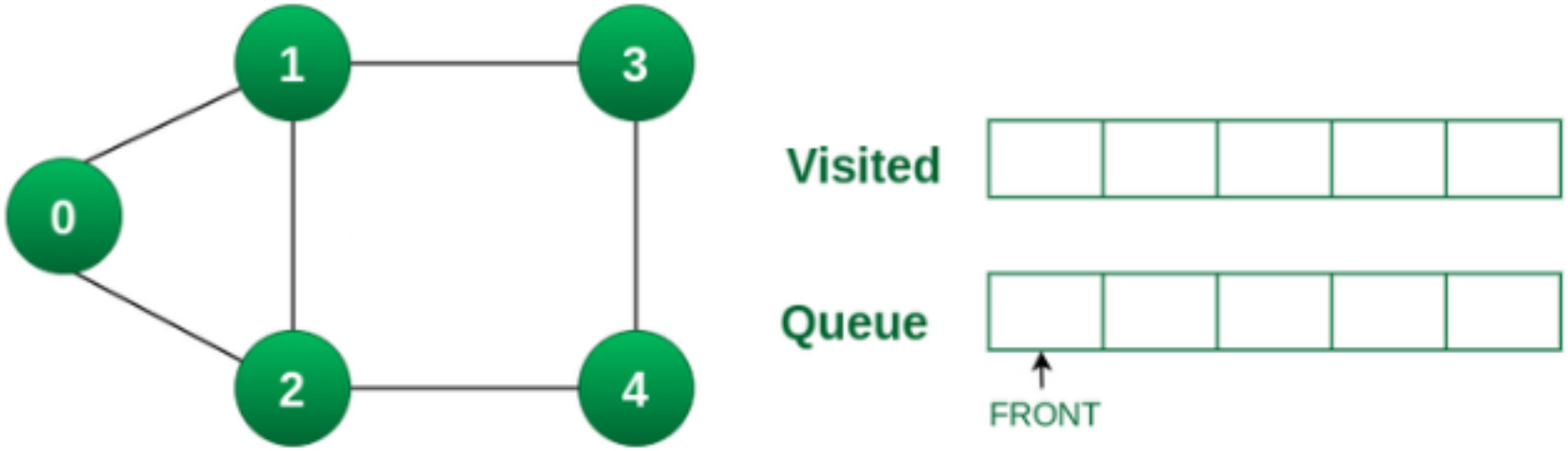
1. The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria.
2. It starts at the root (a given vertex) of the graph and visits/checks/processes all nodes at the current depth level before moving on to the nodes at the next depth level.
3. To avoid processing a node more than once, we divide the vertices into two categories:
   - Visited,
   - Not visited.

BFS

1. A Boolean Visited array is used to mark the visited vertices.
2. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
3. BFS uses a queue data structure for traversal.

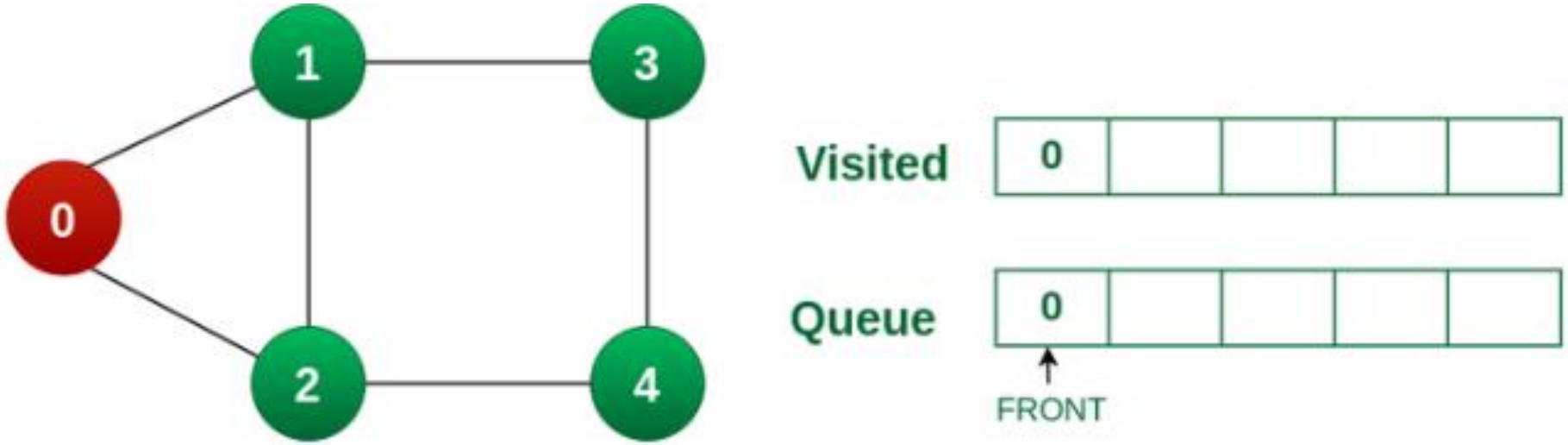
BFS - Steps

1. Initially queue and visited arrays are empty.



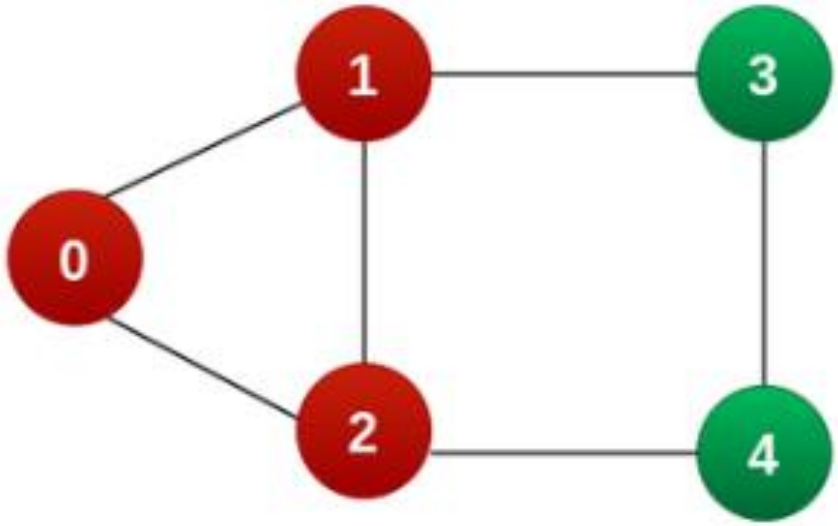
BFS - Steps

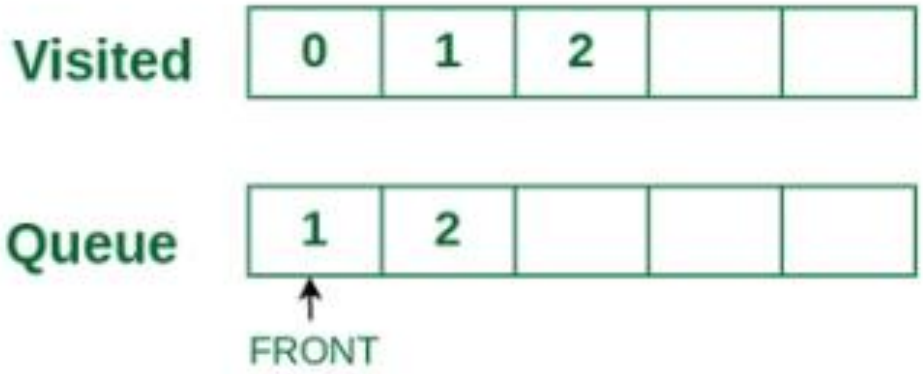
1. Enqueue node 0 into queue and mark it visited.



BFS - Steps

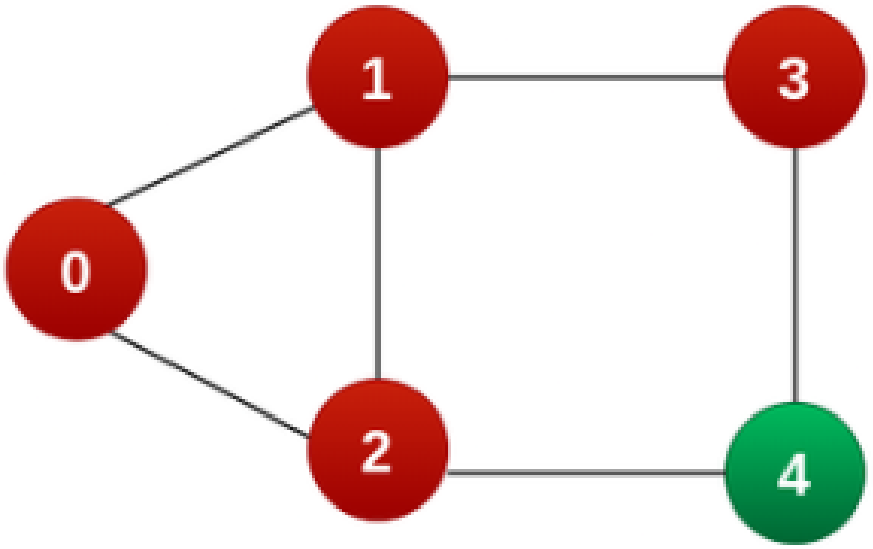
1. Remove node 0 from the front of queue and visit the unvisited neighbors of 0 and enqueue them into queue.

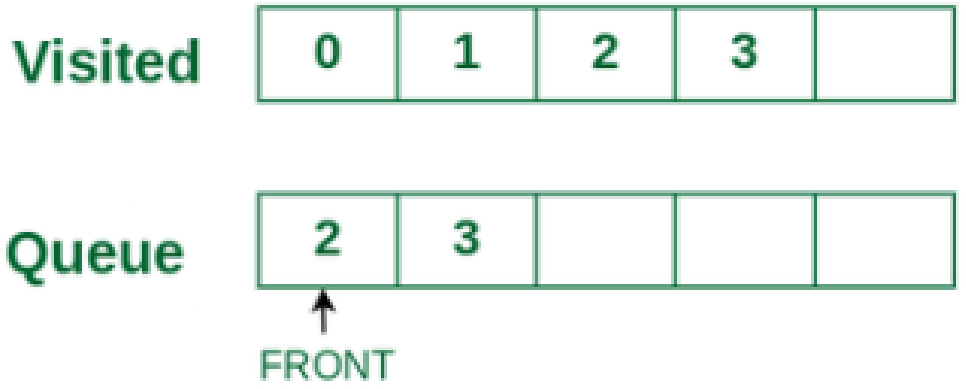




BFS - Steps

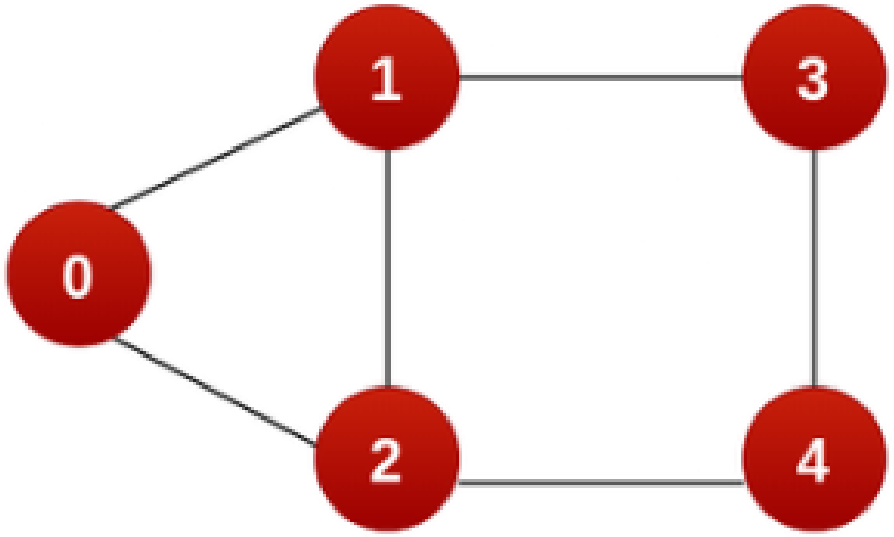
1. Remove node 1 from the front of queue and visit the unvisited neighbors of 1 and enqueue them into queue.

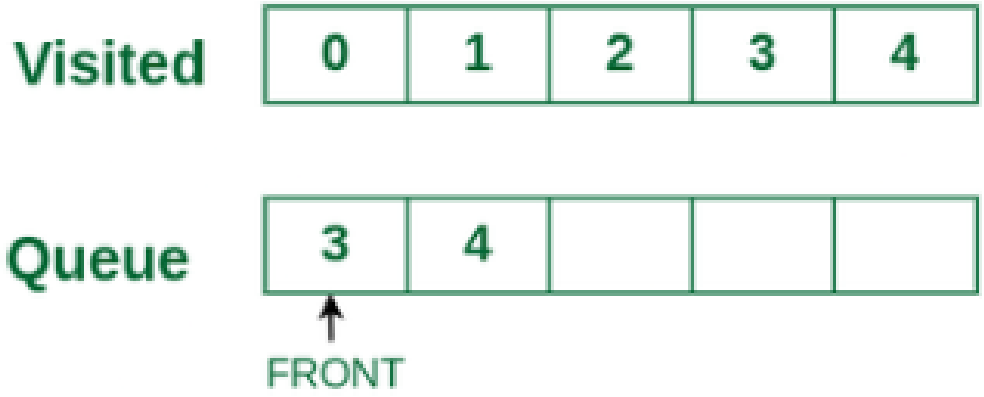




BFS - Steps

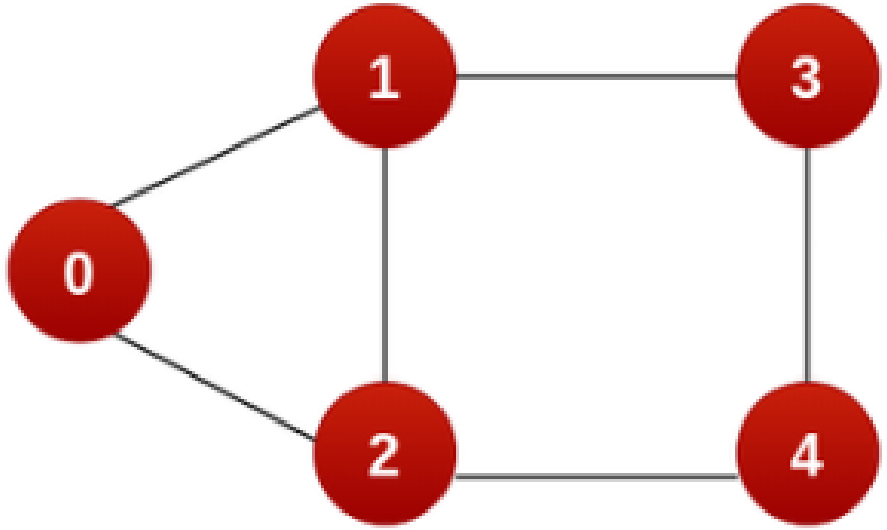
1. Remove node 2 from the front of queue and visit the unvisited neighbors of 2 and enqueue them into queue.

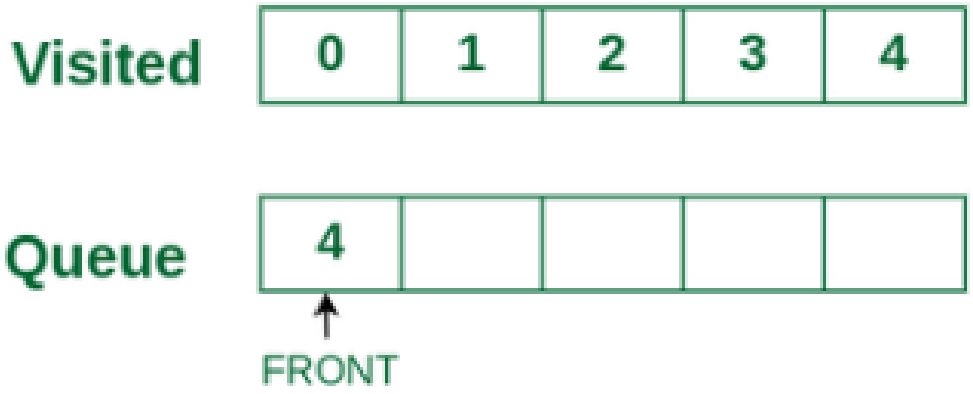




BFS - Steps

1. Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
2. As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.





BFS - Steps

1. Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
2. As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.
3. Now, Queue becomes empty, So, terminate these process of iteration.