JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

DEPARTMENT OF BCA

ANALYSIS AND DESIGN OF ALGORITHMS (16BCA5D11)

Module 3 : Graphs

Prepared by: Prof. Bhavana Gowda D M

Module - 3

GRAPHS

GRAPH:

A **graph** is an abstract data type that is meant to implement the undirected graph and directed graph

- ✓ A graph data structure consists of a finite of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph.

- ✓ These pairs are known as ***edges*, *arcs*, or *lines*** for an undirected graph and as ***arrows*, *directed edges*, *directed arcs*, or *directed lines*** for a directed graph.

GRAPH ABSTRACT DATA TYPE

The graph abstract data type (ADT) is defined as follows:

- **Graph()** : creates a new, empty graph.
- **addVertex(vert) :** adds an instance of Vertex to the graph.
- **addEdge(fromVert, toVert)/(G,X,Y)** : Adds a new, directed edge to the graph that connects two vertices.
- **addEdge(fromVert, toVert, weight)** : Adds a new, weighted, directed edge to the graph that connects two vertices.
- **getVertex(vertKey)** : finds the vertex in the graph named vertKey.
- **getVertices()** : returns the list of all vertices in the graph.

The basic operations provided by a graph data structure *G* usually include :

- **adjacent(*G*, *x*, *y*) :** tests whether there is an edge from the vertex *x* to the vertex *y*.
- **neighbors(*G*, *x*)** : lists all vertices *y* such that there is an edge from the vertex *x* to the vertex *y*;
- **remove_vertex(*G*, *x*) :** removes the vertex *x*, if it is there;
- **remove_edge(*G*, *x*, *y*) :** removes the edge from the vertex *x* to the vertex *y*, if it is there;

- **set_vertex_value(*G*, *x*, *v*)** : sets the value associated with the vertex *x* to *v*.

Structures that associate values to the edges usually also provide:

- **get_edge_value(*G*, *x*, *y*):** returns the value associated with the edge (*x*, *y*)
- **set_edge_value(*G*, *x*, *y*, *v*):** sets the value associated with the edge (*x*, *y*) to *v*.
- **Graph Create()** – return an empty graph
- **Graph InsertNode(Graph, v)** – return a graph with v inserted v has no incident edges.
- **Graph InsertEdge (Graph, v1,v2)** – return a graph with a new edges between v1 & v2
- **Graph DeleteNode (Graph,v)** – return a graph in which v and all edges incident to it are removed.
- **Graph DeleteEdge(Graph,v1,v2)** – return a graph in which the edge (v1,v2) is removed. Leave the incident nodes in the graph.
- **Bool IsEmpty(graph)** – if(graph==empty graph) return True else return False.
- **List Adjacent(graph)** – return a list of all nodes that are adjacent to v.

Graph Representations

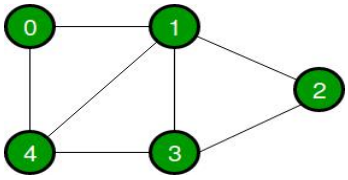
The most commonly used representations of a graph. **.**

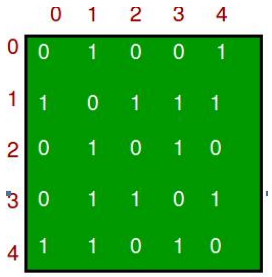
- Adjacency Matrix
- Adjacency List

There are other representations also like, Incidence Matrix and Incidence List

Adjacency Matrix:

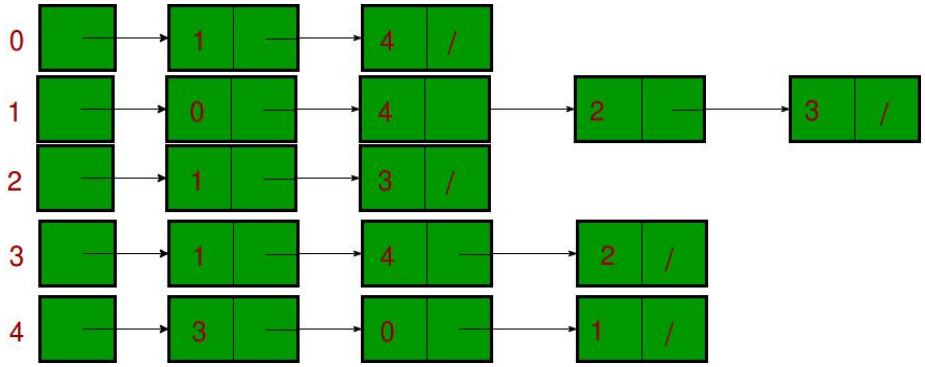
- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.





Adjacency List:

- An array of linked lists is used.
- Size of the array is equal to the number of vertices.
- Let the array be array[].
- An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.
- Following is adjacency list representation of the above graph



Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges.

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image.

Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

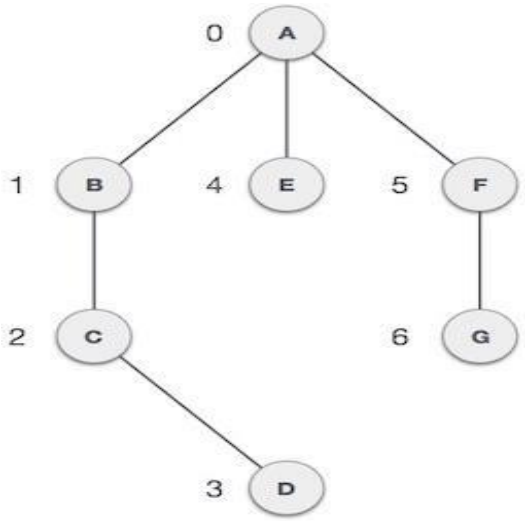
✓ **Vertex** − Each node of the graph is represented as a vertex.

In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

✓ **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

✓ **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

✓ **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations :

Following are basic primary operations of a Graph −

- ✓ **Add Vertex** − Adds a vertex to the graph.
- ✓ **Add Edge** − Adds an edge between the two vertices of the graph.
- ✓ **Display Vertex** − Displays a vertex of the graph.

Graph Types

In addition to simple and weighted descriptions, there two types of graphs:

- ✓ **Directed Graph:** In a directed graph, edges have direction (edges with arrows connect one vertex to another).
- ✓ **Undirected Graph:** In an undirected graph, edges have no direction (arrowless connections). It is basically the same as a directed graph but has bi-directional connections between nodes.

Figure 1 outlines an example of directed and undirected weighted graphs

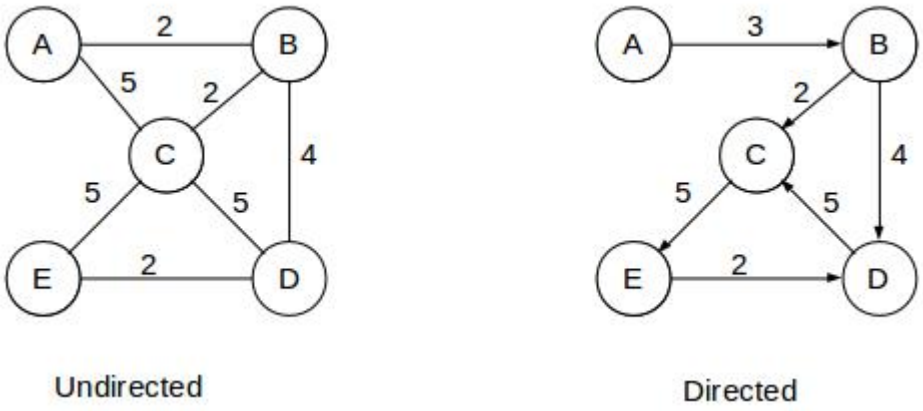


Figure 1: Examples of directed and undirected graphs.

Why Use Graphs?

Using graphs, we can clearly and precisely model a wide range of problems. For example, we can use graphs for:

- Coloring maps (modeling cities and roads)
- Social Relations (sociology)
- Protein interactions (biology)
- Social networking (e.g. Facebook and Twitter)

Graph Traversals :

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

Traversal techniques are :

1. Depth First Search (DFS) and
2. Breadth First Search (BFS)

1. Depth First Search

- ✓ The DFS algorithm is a recursive algorithm that uses the idea of backtracking.
- ✓ It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
- ✓ Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.
- ✓ All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.
- ✓ Data Structure used is : **Stack**

The basic idea is as follows:

- ✓ Pick a starting node and push all its adjacent nodes into a stack.
- ✓ Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- ✓ Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked.
- ✓ This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm DFS(u,n,a)

Step 1: [Visit the vertex u]

S[u]<-1

Step 2:[Traverse deeper into the graph till we get the dead end or till all vertices are visited

for v<-0 to n-1 do

if(a[u][v]=1 ands[v]=0) **then**

DFS(v,n,a)

end if

end for

Step 3:[store the dead vertex or which is completely explored]

j<-j+1

res[j]<-u

Step4:[Finished]

return //End of DFS Algorithm