Graph:

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

Representations of Graph:

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbours of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use

Here are the two most common ways to represent a graph:

1.Adjacency Matrix
2.Adjacency List

1. Adjacency Matrix:

   Adjacency matrix is a sequential representation.

   It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.

   In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, ai,j = 1, otherwise ai,j= 0.

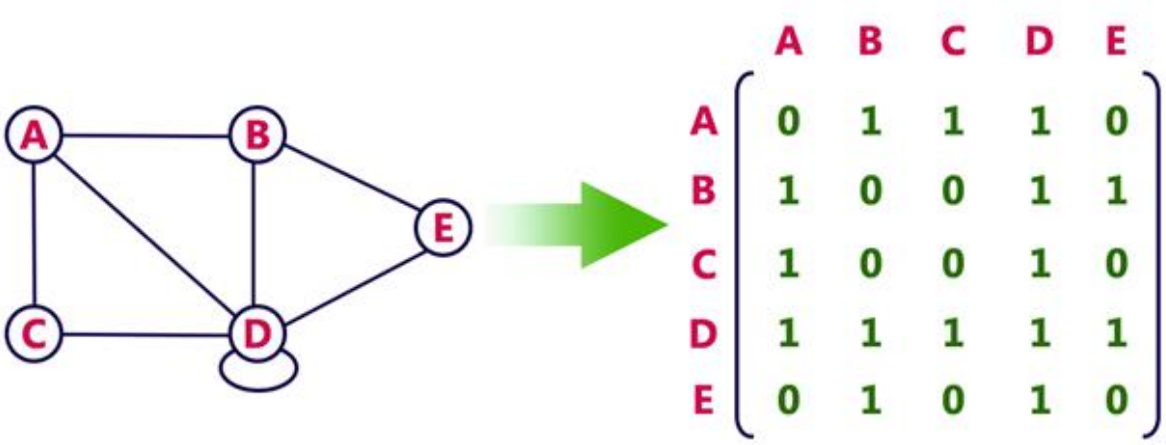
   If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

   An adjacency matrix is a way of representing a graph as a matrix of Boolean (0’s and 1’s).

Let’s assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n.

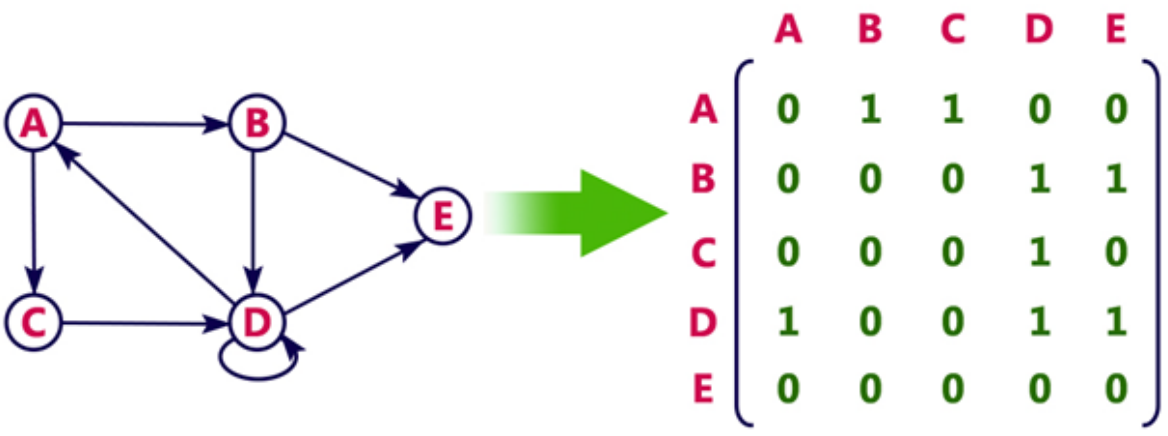
If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
If there is no edge from vertex i to j, mark adjMat[i][j] as 0.

Undirected graph representation

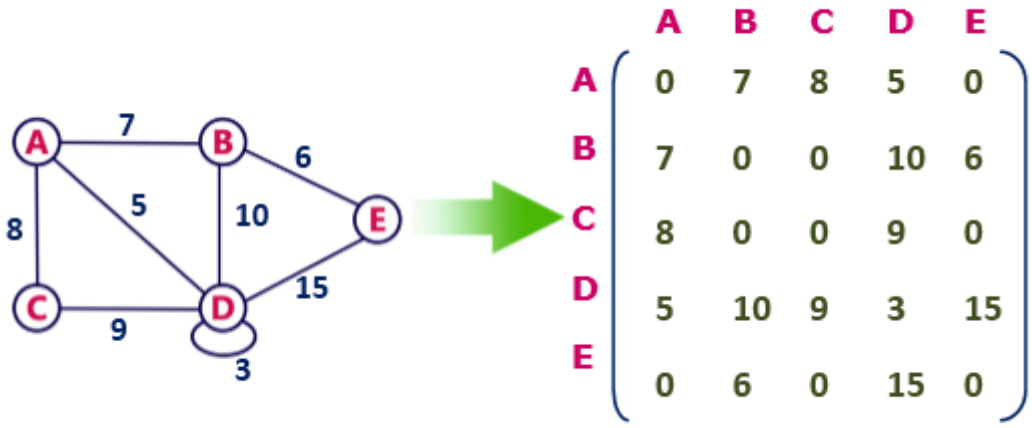


Directed graph representation

See the directed graph representation:



Undirected weighted graph representation:



Cons: It takes a lot of space and time to visit all the neighbours of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

2. Incidence Matrix

In **Incidence matrix representation**, graph can be represented using a matrix of size:

Total number of vertices by total number of edges.

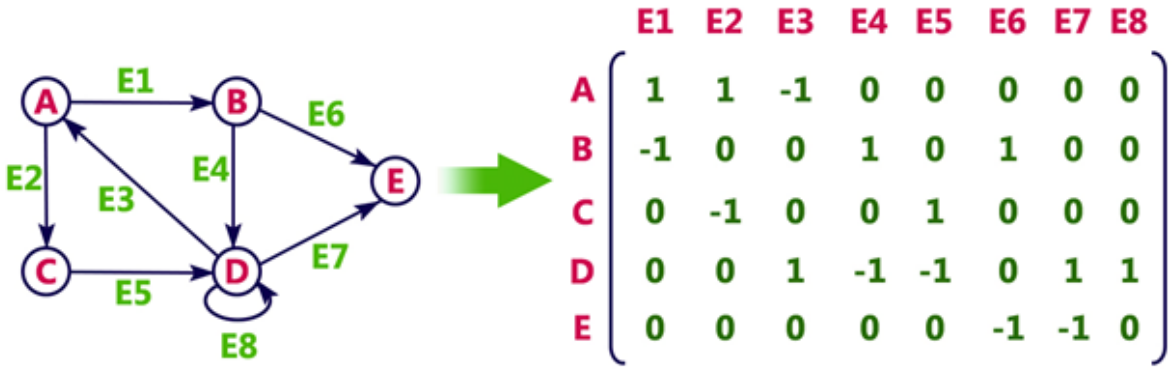
It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.

This matrix is filled with either **0 or 1** or -1.

- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

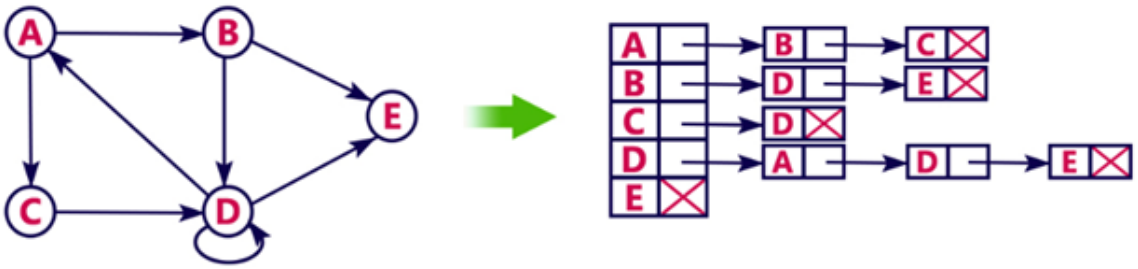
Example

Consider the following directed graph representation.

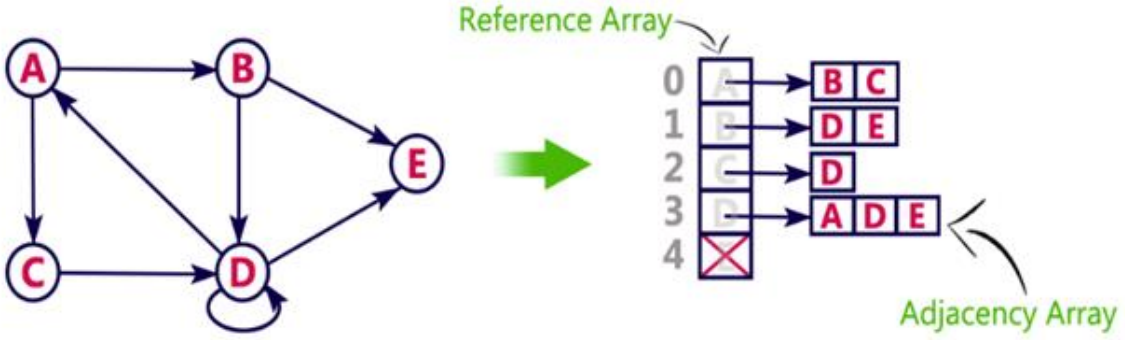


3. Adjacency List

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbours. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbours of v.



We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Cons:

The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

Graph traversal is a process of visiting all the vertices of a graph. It can be done in two ways: depth-first search (DFS) and breadth-first search (BFS).

Depth-first search (DFS) is a recursive algorithm that starts at a given vertex and explores as far as possible along each branch before backtracking. DFS can be used to find all the vertices in a graph, or to find a path between two vertices.

Breadth-first search (BFS) is an iterative algorithm that starts at a given vertex and explores all the neighbouring vertices before moving on to the next level. BFS can be used to find the shortest path between two vertices, or to find all the vertices within a given distance of a vertex.