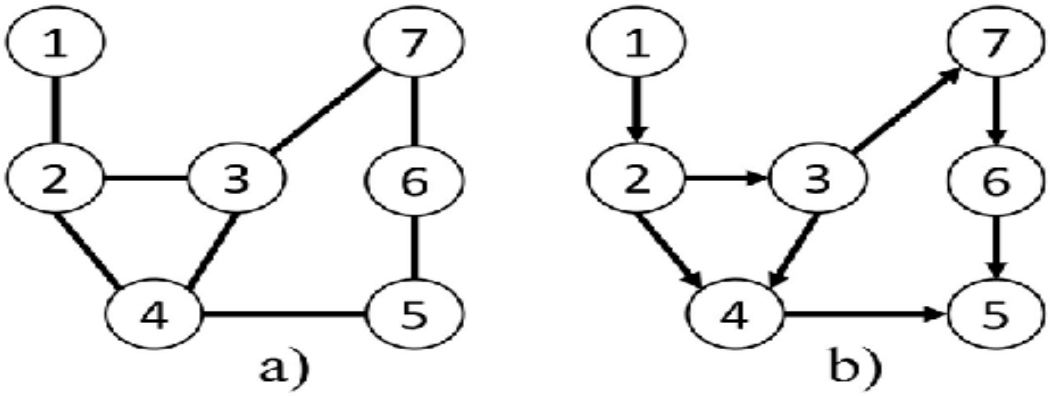
Lab Cycle: 9 **CSL 201 Data structures Lab**

Date of Submission: 4-3-2022

Faculty In charge: Dr Binu V P

Objective: Learn Graph

Graph is a mathematical structures used to model relations between objects from a certain collection. The interconnected objects are represented by mathematical abstractions called vertices, and the links that connect some pairs of vertices are called edges. Typically, a graph is depicted in diagrammatic form as a set of circles for the vertices, joined by lines or curves for the edges graph may be undirected(a), meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another called directed graph(b).



Graph Representations

A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented

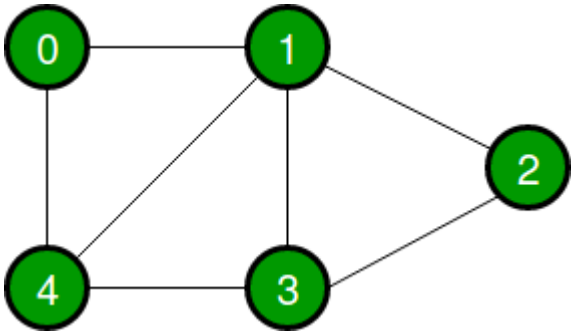
with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale

There are different ways to store graphs in a computer system. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. Theoretically one can distinguish between list and matrix structures but in concrete applications the best structure is often a combination of both. List structures are often preferred for sparse graphs as they have smaller memory requirements. Matrix structures on the other hand provide faster access for some applications but can consume huge amounts of memory.

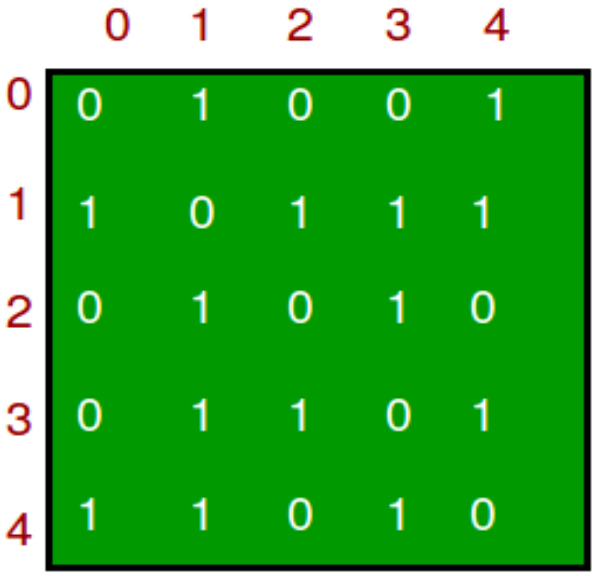
Two main data structures for the representation of graphs are used in practice. The first is called an adjacency list, and is implemented by representing each node as a data structure that contains a list of all adjacent nodes. The second is an adjacency matrix, in which the rows and columns of a two-dimensional array represent source and destination vertices and entries in the array indicate whether an edge exists between the vertices. Adjacency lists are preferred for sparse graphs; otherwise, an adjacency matrix is a good choice.

AdjacencyMatrix:

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.



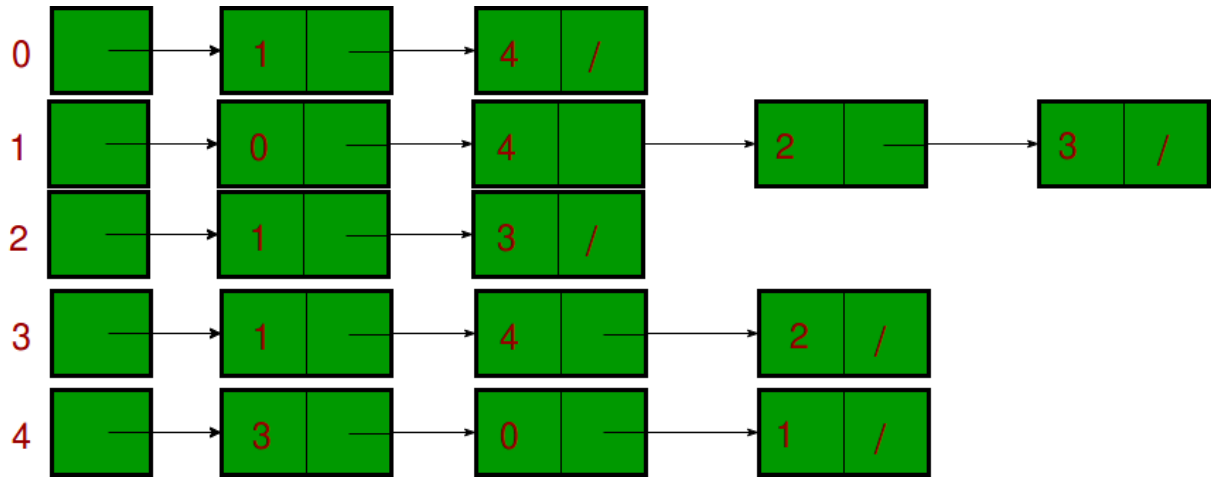
The adjacency matrix for the above example graph is:



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

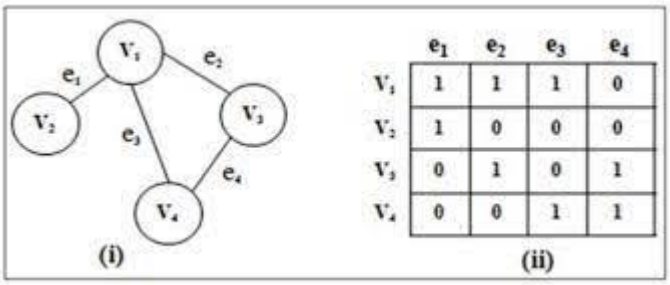
Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).
Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

AdjacencyList:
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the 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 represented as lists of pairs. Following is the adjacency list representation of the above graph.



Incidence Matrix

The incidence matrix A of an undirected graph has a row for each vertex and a column for each edge of the graph. The element A[[i,j]] of A is 1 if the i^{th} vertex is a vertex of the j^{th} edge and 0 otherwise.



	e_1	e_2	e_3	e_4
V_1	1	1	1	0
V_2	1	0	0	0
V_3	0	1	0	1
V_4	0	0	1	1

Graph Algorithms

Graph algorithms are a significant field of interest within computer science. Typical operations associated with graphs are: finding a path between two nodes, like depth-first search (DFS) and breadth-first search(BFS)

Depth-first search (DFS)

Depth-First Search (DFS) is an algorithm for traversing or searching a tree, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for exploration.

There are various ways to traverse (visit all the nodes) of a graph systematically. A couple of these ways (depth-first and breadth-first) give us some information about graph structure (e.g. connectedness).

In depth-first search the idea is to travel as deep as possible from neighbor to neighbor before backtracking. What determines how deep is possible is that you must follow edges, and you don't visit any vertex twice.

To do this properly we need to keep track of which vertices have already been visited, plus how we got to (the path to...) where we currently are, so that we can backtrack. We could keep track of which nodes were visited in a Boolean array, and a stack to push nodes onto that we mean to visit (the course Readings have a recursive algorithm for DFS which takes a slightly different approach). Here's some pseudo code:

```
DFS(G,v)   ( v is the vertex where the search starts )
   Stack S := {};   ( start with an empty stack )
   for each vertex u, set visited[u] := false;
   push S, v;
   while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
         visited[u] := true;
         for each unvisited neighbour w of u
            push S, w;
      end if
   end while
END DFS()
```

It would probably be useful to keep track of the edges we used to visit the vertices, since these edges would span the vertices visited. One way to do this is with another array predecessor[u] which indicates which vertex u was reached from. When we are processing the neighbours of, say, vertex u, for each neighbour (say v) of u that we push onto the stack, we set

predecessor[v] to u. Eventually we end up with a tree: an acyclic, connected graph of all the vertices that can be reached from our starting point.

What happens if our original graph G isn't connected? Then DFS(G,v) won't visit any vertices that aren't connected to its starting point. You'll need an outer loop that iterates over unvisited vertices, and then calls DFS(G,V).

The end result is a forest (a collection of trees) representing the connected components of G.

BFS

In graph theory, Breadth-First Search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Algorithm (informal)

- Enqueue the root node.
- Dequeue a node and examine it.
- If the element is found in this node, quit the search and return a result.
- Otherwise enqueue any successors (the direct child nodes) that have not yet been examined.
- If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
- Repeat from Step 2.

Note: Using a stack instead of a queue to store the nodes yet to be visited would turn this algorithm into a depth-first search.

Space complexity of DFS is much lower than BFS (breadth-first search). It also lends itself much better to heuristic methods of choosing a likely-looking branch. Time complexity of both algorithms are proportional to the number of vertices plus the number of edges in the graphs they traverse (O(|V| + |E|)).

Implement The Graph representations

1.Adjacency Matrix

2.Incident Matrix

3.Adjacency List

Answer the queries like degree of a vertex. List of vertices adjacent to a vertex. Etc..

Implement BFS and DFS