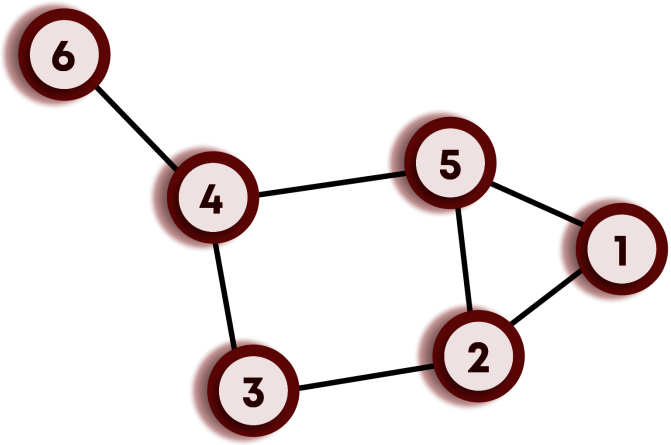
Graph 1

Introduction

A **graph** is a pair $G = (V, E)$, where V is a set whose elements are called ***vertices***, and E is a set of two sets of vertices, whose elements are called *edges*.

The vertices x and y of an edge $\{x, y\}$ are called the ***endpoints*** of the edge. The edge is said to ***join*** x and y and to be ***incident*** on ***x*** and ***y***. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use Graphs to store it. In such cases, we will be using graphs. Refer to the figure for a better understanding.

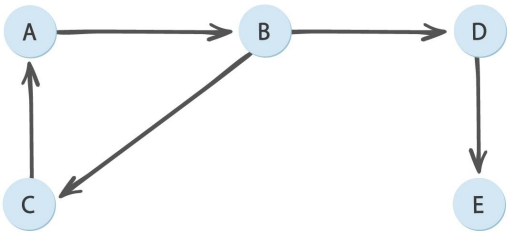


Graphs Terminology

- Nodes are named **vertices,** and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be **(N-1)**, where **N** is the number of nodes.
- In a complete graph (where each node is connected to every other node by a direct edge), there are $^{N}C_{2}$ number of edges means **(N * (N-1)) / 2** edges, where n is the number of nodes.
- This is the maximum number of edges that a graph can have.
- Hence, if an algorithm works on the terms of edges, let's say **O(E)**, where **E** is the number of edges, then in the worst case, the algorithm will take **$O(N^2)$** time, where **N** is the number of nodes.

Graphs Representation

Suppose the graph is as follows:

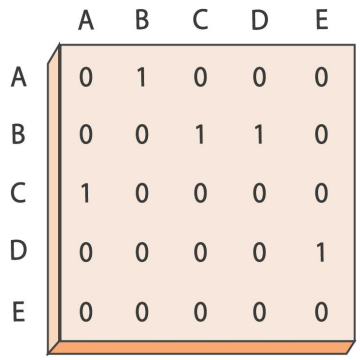


There are the following ways to implement a graph:

1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to $O(n^2)$ time complexity in the worst case. Pictorial representation for the above graph using the edge list is given below:

2. **Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check in its list if the target node is present or not. This will take O(n) work to figure out a particular edge.
3. **Adjacency matrix:** Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each

node stores only those nodes that are directly connected to them. For the above graph, the adjacency matrix looks as follows:



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

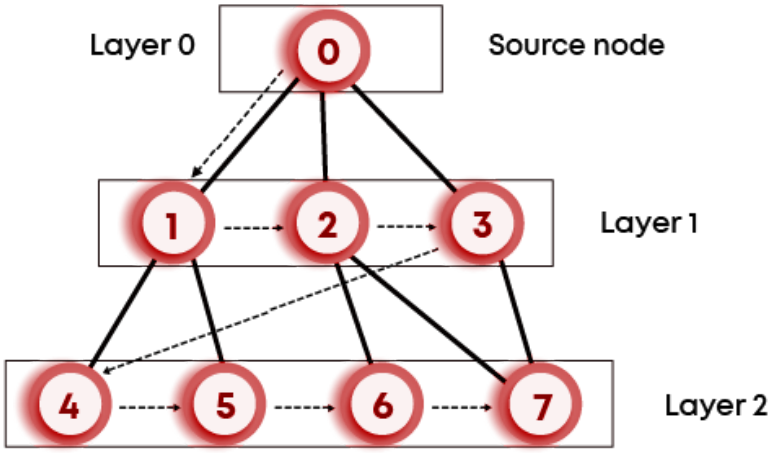
Graph traversal techniques

BFS Traversal:

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the Graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

Algorithm:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



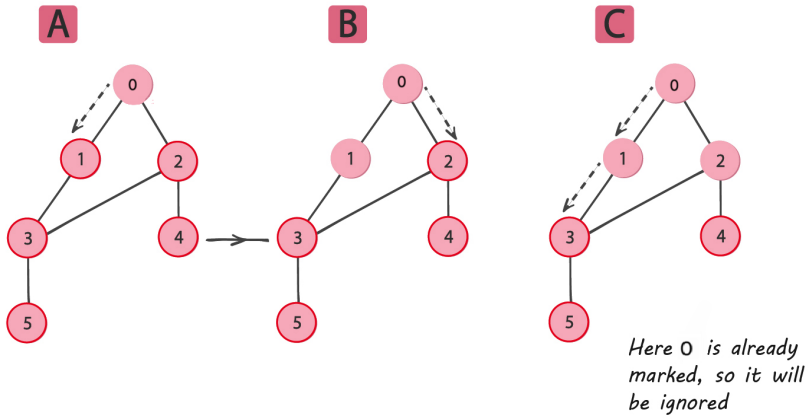
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

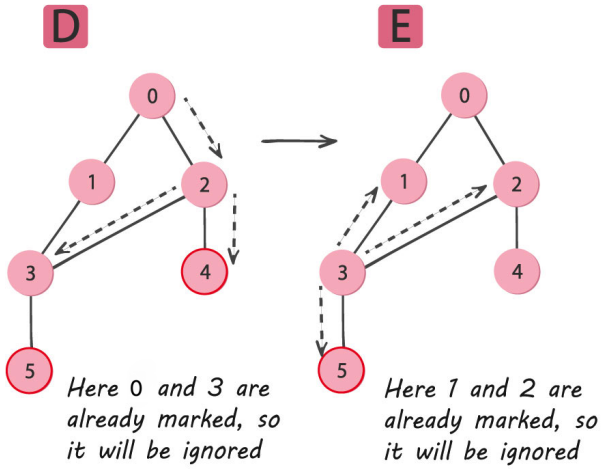
Code:

```
vector<int> ar[100001];
bool vis[100001];
int dist[100001];

void bfs(int src){
      queue<int> q;
      q.push(src);
      dist[src] = 0;
      vis[src] = true;

      while(q.empty() == false){
            int node = q.front();
            q.pop();

            cout<<node<<" ";
```

```
            for(int v : ar[node]){
                  if(vis[v] == false){
                        dist[v] = dist[node] + 1;
                        vis[v] = true;
                        q.push(v);
                  }
            }
      }
}
```

Consider the dry run over the example Graph below for a better understanding of the same:





DFS Traversal:

Depth-first search (DFS) is an algorithm for traversing or searching Graph or Graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a Graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Algorithm:

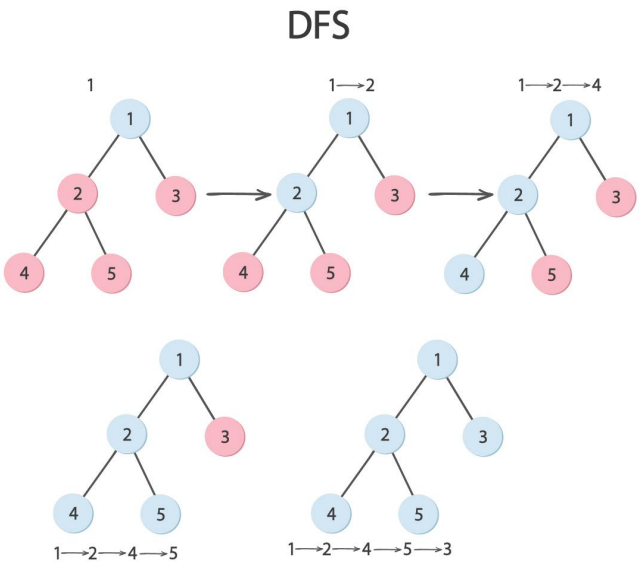
- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Code:

```
void dfs(int node){
      vis[node] = true;
      cout<<node<<" ";
      for(int v : ar[node]){
            if(vis[v] == false) dfs(v);
      }
}
```

Time complexity: **O(V + E)**, where **V** is the number of vertices and **E** is the number of edges in the Graph.

Space Complexity: O(V), since an extra visited array of size **V** is required.



Has Path

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Approach:

This can be simply solved by considering the vertex v1 as the starting vertex and then running either BFS or DFS as per your choice, and while traversing if we reach the vertex v2, then we will simply return true, otherwise, return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Detect Cycle in a Directed Graph

Problem Statement: Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false.

Approach:

To change a directed tree to a cyclic directed graph by adding one edge, we can choose a node that has more than zero ancestors and add an edge from that node to one of its ancestors. This can always form a complete cycle. This edge is called the back edge.

The basic idea is to traverse a graph using **DFS** and keep track of visited nodes and search for the back edge. If we found any back edge, we will return **true**, else we will return **false**.

DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by **DFS**.

To detect a back edge, we can use coloring to keep track of visited nodes as well as ancestors. The color **0** represents an unvisited node. All the nodes with the color **1** represent the ancestor of the node. The color **2** represents visited nodes that are not ancestors of the node. We can traverse the graph using DFS and for each node, we check if there is any back edge that connects to its ancestor with color **1**. If we find any back edge then return true otherwise return **false**.

For a disconnected graph, we run the same algorithm of each connected graph and return true if there is a back edge in any of the connected graphs.

Algorithm:

- Insert all the edges in the graph and make an adjacency list **graph**.
- Initialize an array **color** with all the values **0** representing unvisited nodes.
- Iterate over the nodes and for each unvisited node with color **0**: traverse the graph using **DFS**. If **DFS** returns true then there is a back edge in the graph. So return **true**.
- Consider the recursive function **DFS** that takes three arguments, **U** represents the node number, **color** represents the color of the nodes, the **graph** represents the adjacency list.
- In **DFS**, first color the node **U** with **1**, which represents the node **U** will be the ancestor of the future nodes.
- Iterate over all adjacent nodes **V**:

- If the color of node **V** is 1, that means **V** is the ancestor of **U**, which also means the edge from **U** to **V** is the back edge which forms a cycle. So return **true**.
- If the color of node **V** is **0**, make a recursive call on node **V** to check for a back edge in the subtree of **V**. if there is a back edge then return true.
- If there we can not find the back edge, return **false**.
- Otherwise, return **false**.

Code:

```
bool DFS(int u, vector<int>& color, vector<vector<int>>& graph)
{
	color[u] = 1;
	for (int i = 0; i < graph[u].size(); i++)
	{
		int v = graph[u][i];
		if (color[v] == 1)
		{
			return true;
		}
		if (color[v] == 0 && DFS(v, color, graph))
		{
			return true;
		}
	}
	color[u] = 2;
	return false;
}
bool isCyclic(vector<vector<int>>& edges, int v, int e)
{
	vector<vector<int>> graph(v + 1);
	for (int ei = 0; ei < e; ei++)
	{
		int a = edges[ei][0];
		int b = edges[ei][1];
		graph[a].push_back(b);
	}
```

```
        vector<int> color(v, 0);
        for (int i = 0; i < v; i++)
        {
                if (color[i] == 0 && DFS(i, color, graph) == true)
                {
                        return true;
                }
        }
        return false;
}
```

Time Complexity: O(V + E), Where **V** is the number of nodes and **E** represents the number of edges in the given graph.

Since we are visiting each node and edge of the given graph only once. Therefore time complexity will be **O(V + E)**.

Space Complexity: O(V + E), Where **V'** is the number of nodes and **E** represents the number of edges in the given graph.

Since we are constructing an adjacency list and storing the color of each node in an array of size **N**. So the overall space complexity will be **O(V + E)**.

Topological Sorting

Problem Statement: Given a DAG(direct acyclic graph), print Topological Sorting of a given graph.

Topological Sort: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v

in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Approach:

We can modify DFS to find the Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print the contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Code:

```
#include<bits/stdc++.h>
using namespace std;

const int N = 100000;
vector<int> ar[N+1];
int inDegree[N+1];

void topSort(int n){

	queue<int> q;

	for(int i=1;i<=n;i++)
		if(inDegree[i] == 0) q.push(i);

	while(q.empty() == false){
		int node = q.front();
		q.pop();
```

```
            cout<<node<<" ";

            //node -> v
            for(int v : ar[node]){
                  inDegree[v]--;

                  if(inDegree[v] == 0) q.push(v);
            }

      }

}
int main(){
      int n, m, a, b;

      cin>>n>>m;

      for(int i=1;i<=n;i++) inDegree[i] = 0;

      while(m--){
            cin>>a>>b;
            ar[a].push_back(b);
            inDegree[b]++;
      }

      topSort(n);
}
```

Time Complexity: O(V+E), Where **V** is number of nodes and **E** is number of edges.

The above algorithm is simply DFS with an extra stack. So time complexity is the same as of DFS.

Space Complexity: O(V), Where **V** is number of nodes.

The extra space is needed for the stack.