Data Structure

6. Graphs

Handwritten by pankaj kumar

GRAPH

Date
Page No. 296.

(1) Introduction :-

- A graph 'G' is simply a way of encoding pairwise relationships among a set of objects
- the graph consist of V of nodes and E of edges, each edges joins two nodes. are, $G = (V, E)$.
- if a edge $e \in E$ connect two node $(u, v) \in V$, then we can represent them as $e = \{u, v\}$, u & v are end of e.

* Some important definitions

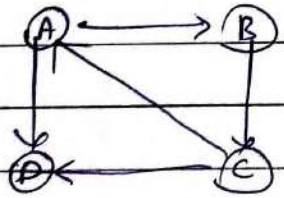
- **Direct edge :-**
  (i) ordered pair of vertices (u, v)
  (ii) First vertex u is origin
  (iii) second vertex v is the destination

- asymmetric relationship b/w end.

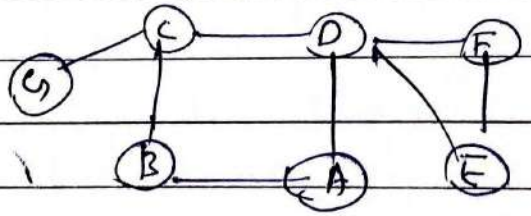
- **Undirected edge :-**
- Unordered pair of vertices (u, v)
- ex :- rail/way lines.

- symmetric relationship b/w end.

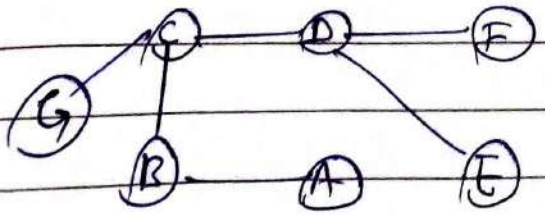
- **Directed graph :-**
- All the edges are directed
- Ex :- route network



- **Undirected graph :-**
- All the edges are undirected
- Ex :- Flight network

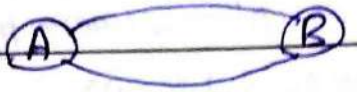


- when the edge connects two vertices, the vertices are said to be 'adjacent' to each other and the edge is incident on both vertices.
- A Graph with no cycles is called a tree. A tree is an acyclic connected graph.

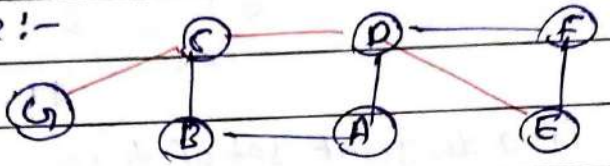


- A self loop is an edge that connects a vertex to itself.

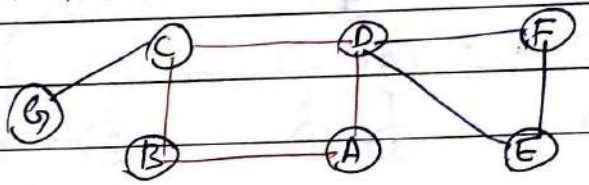
- Two edges are parallel if they connect the same pair of vertices.



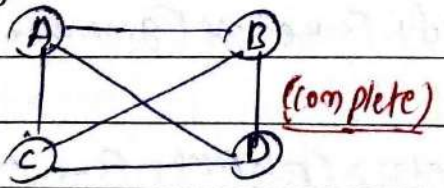
- The degree of a vertex is the no. of edges incident on it.
- A path in a graph is a sequence of adjacent vertices. simple path is a path with no repeated vertices. paths from G to E are :-



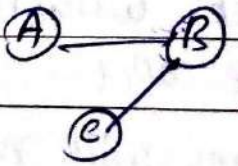
- A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).



- We say that one vertex is connected to another if there is a path that contains both of them. in case of directed there should be ($v \to u$ & $u \to v$ both).
- A graph is connected if there is a path from every vertex to every other vertex. in case of directed ($u \to v$ & $v \to u$ both)



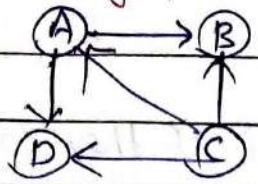
(complete)



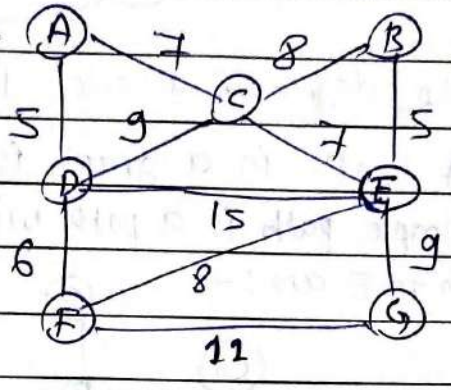
(connected)

- Graph with all edges present are called complete graph. i.e graph of each node u is adjacent to all other nodes v of 'G'.
- A complete graph is always a connected graph.

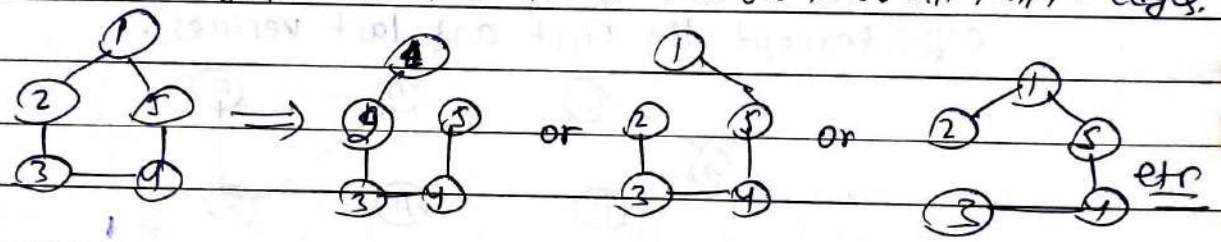
- A directed acyclic graph [DAG] is a directed graph with no cycle.



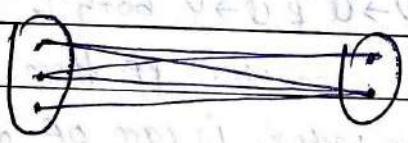
- In weighted graphs integers (weights) are assigned to each edge to represent (distance) or, (costs).



- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. with minm edges.



- A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



- Graph with relatively Few edges (generally if edges $< |V| \log |V|$) are called sparse graphs.
- Graph with maxm edges (relatively Few of the possible edges missing) are called dense graphs.
- Directed weighted graph sometimes called network.
- If no. of vertices of a graph is $|V|$. and no. of edges ~~the~~ $|E|$. the $|E|$ can be. in undirected graphs.

$$|E| \leq \frac{|V|(|V|-1)}{2}$$

(2) Applications of Graphs

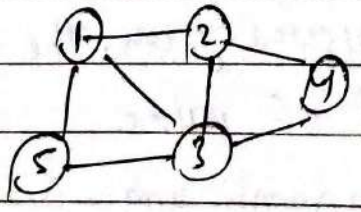
- Representing relationships b/w components in electronic circuits.
- Transportation network: Highway network, Flight network.
- Computer network: Local Area Network, Internet, Web.
- Database: For representing ER (entity Relationship) diagram, for representing dependency of table.
- Operating System: Resource allocation Graph,
- Microsoft Excel: uses DAG. (Directed Acyclic graphs)
- Social media sites: uses graph to track the data of users. liked showing preferred post suggestions. recommendation.

(3) GRAPH Representation

There are 3 ways to represent graph
- 'Adjacency Matrix' - 'Adjacency list' - Adjacency set.

(1) Adjacency Matrix:

- adjacency matrix is a 2D array of size V×V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j]=1 indicate 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 j with weight w.
- For undirected graph adj[i][j] = adj[j][i] = edge i to j or j to i
  For directed graph adj[i][j] ≠ adj[j][i], if edge is i to j only adj[i][j] will be consider for adj[j][i] there should also be edge j to i



⇒

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	0	1
2	0	1	0	1	1	0
3	0	1	1	0	1	1
4	0	0	1	1	0	0
5	0	1	0	1	0	0

Code:-

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    int adj[n+1][n+1];
    for(int i=0; i<m; i++){
        int u,v;
        cin>>u>>v;
        adj[u][v] = 1;
        adj[v][u] = 1;
    }
    return 0;
}
```

For weighted graph :-

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    int adj[n+1][n+1];
    for(int i=0; i<m; i++){
        int u,v,w;
        cin>>u>>v>>w;
        adj[u][v]=w;
        adj[v][u]=w;
    }
    return 0;
}
```

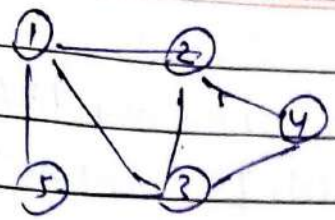
here, n = no. of vertices, m = no. of edges, w = weight of (u,v)

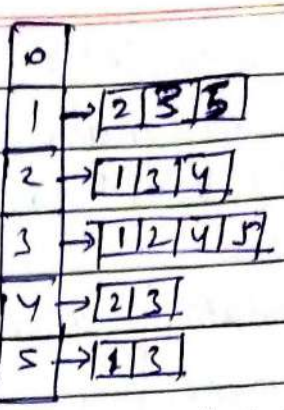
Pros:- • Representation is easy,
• Removing an edge take O(1) time,
• To find edge is present or not take O(1) time)

Cons:- • Consumes more space $O(V^2)$. Even if the graph is sparse (contains less no. of edges), consumes same space.
• Adding a vertex take $O(V^2)$ time.
• Computing all neighbours of a vertex taker O(V).

(2) Adjacency List:-

• an array of lists is used. The size of array is equal to the no. of vertices.
• Let the array be an adj[]. An entry adj[i] represent the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weight of edges can be represent as the list of pairs.
• a list means linked list or dynamic array (vector).





Code:-

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n+1];
    for(int i=0;i<m;i++){
      int u,v;
      cin>>u>>v;
      adj[u].push_back(v);
      adj[v].push_back(u);
    }
    return 0;
}
```

For weighted graph:-

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    vector<pair<int,int>> adj[n+1];
    for(int i=0;i<m;i++){
      int u,v,w;
      cin>>u>>v>>w;
      adj[u].push_back({v,w});
      adj[v].push_back({u,w});
    }
    return 0;
}
```

print graph:-

```
void printGraph(vector<int> adj[], int n){
    for(int v=0;v<n+1;v++){
        cout<<v;
        for(auto x: adj[v]) cout<<"->"<<x;
        cout<<endl;
    }
}
```

Pros:- • space O(|V|+|E|)
• adding a vertex is easier
• computing all neighbour of a vertex takes optimal time.

Cons:- • To find edge present or not take O(v) time

Note:- In real life graph is sparse (|E|<<|V|²). that's why adjacency list DS is commonly used for storing graphs.

(3) Adjacency Set :-

- it is very much similar to adjacency list but instead of using linked list, Disjoint sets [Union-Find] are used.

Notes :-

Representation	Space	checking edge b/w u & v	iterate edges incident to u
Adj. matrix	V^2	1	$O(V)$
Adj list	$E+V$	$O(V)$	$O(V)$
Adj Set	$E+V$	$O(\log V)$	$O(V)$

(4) GRAPH Traversal :-

- Graph traversal is also called graph search algorithms. We can thought as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. There are two search algorithm.
  - * Depth First Search [DFS]
  - * Breadth First Search [BFS]

(1.) Depth First Search [DFS] :-

- In a DFS, we go as deep as possible down one path before backing up and trying a different one.
- It is similar to preorder traversal of the trees. The only catch here is, graph may contain cycles (a node maybe visited twice). To avoid processing a node more than once, use a boolean visited array.

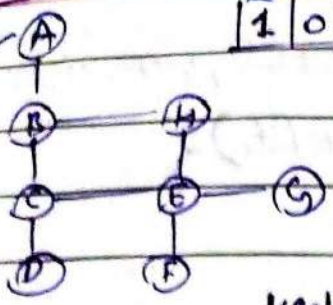
Approach :-

1. Start with a random node from graph
2. make an array to keep track visited node, once visited make the node as visited.
3. print this current node
4. Get neighbour nodes and perform above steps recursively for each node deeply/depthly if node is unvisited.

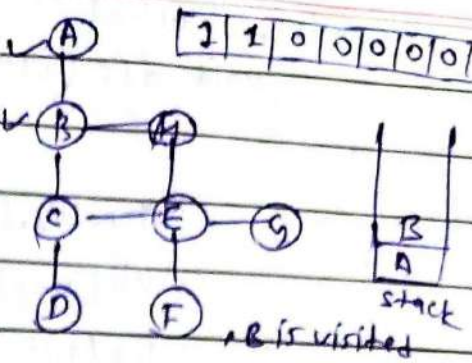
Ex :-

visited table

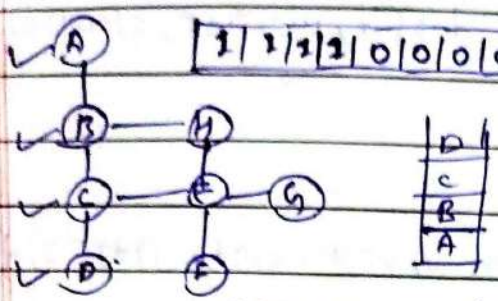
1	0	0	0	0	0	0	0



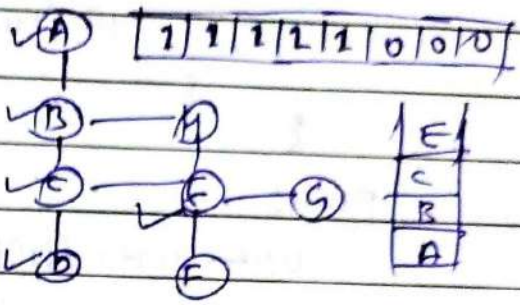
Vertex A is visited



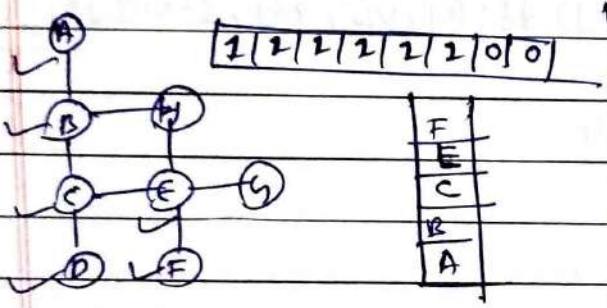
B is visited



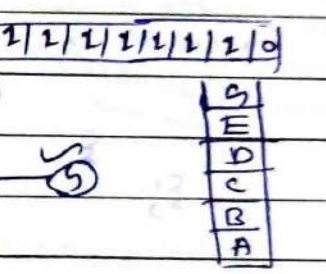
Now, no node is unvisited adjacent of D



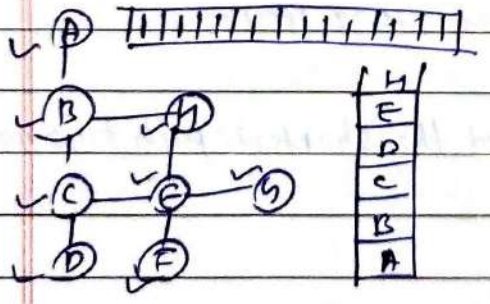
Now, D will be popout and now C is on top there is E which is 'c' adjacent & unvisited so, it will called



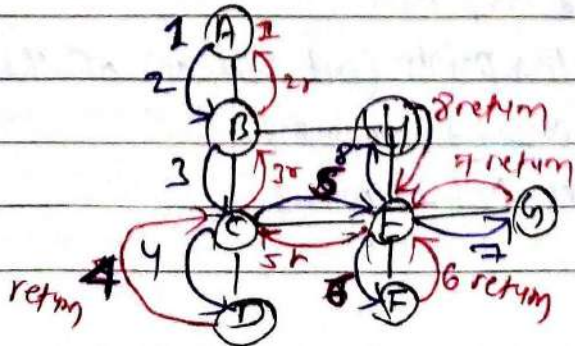
Now there is no any adjacence of F which is unvisited so backtrack to 'E' & goes to G, E's adjacence.



Back track from G to E & goes to H.



Now, there is no unvisited which is adjacence to H so back track to E, again, E → D, D → C, C → B → A, we will now out the first recursion call. finaly stack will be empty.



O/P :- A B C D E F G H

```
class Solution{
    void dfs(int node, vector<int>& vis, vector<int> adj[],
                    vector<int>& storeDfs){
        storeDfs.push_back(node);
        vis[node]=1;
        for(auto it : adj[node]){
            if(!vis[it]) dfs(it, vis, adj, storeDfs);
        }
    }
public:
    vector<int> dfsOfGraph(int V, vector<int> adj[]){
        vector<int> storeDfs;
        vector<int> vis(V+1,0);
        for(int i=1; i<=V; i++){
            if(!vis[i]) dfs(i, vis, adj, storeDfs);
        }
        return storeDfs
    }
};
```

This will also work for disconnected graph bcz we are traversing all node

connected component in a graph

Time: O(N+E), N= time taken for visiting N Nodes and E for travelling through nodes.

Space: O(N+E) + O(N) + O(N).

advantages:-

- DFS on Binary tree generally take less memory than BFS
- It can be easily implemented using recursion

disadvantages:-

- a DFS doesn't necessarily find the shortest path to a node, while BFS does

Application:-

- Topological sort
- Finding connected component
- Finding articulation points (cut vertices) of the graph.
- Finding strongly connected component
- solving puzzles such as mazes.

② Breadth First Search [BFS] :-

- In a BFS we first explore all the nodes one step away, then all the nodes two steps away, etc. it is like throwing a stone in pond. the nodes we explore "ripple out" from the starting point.
- It works similar to level-order traversal of trees. it also uses queues.

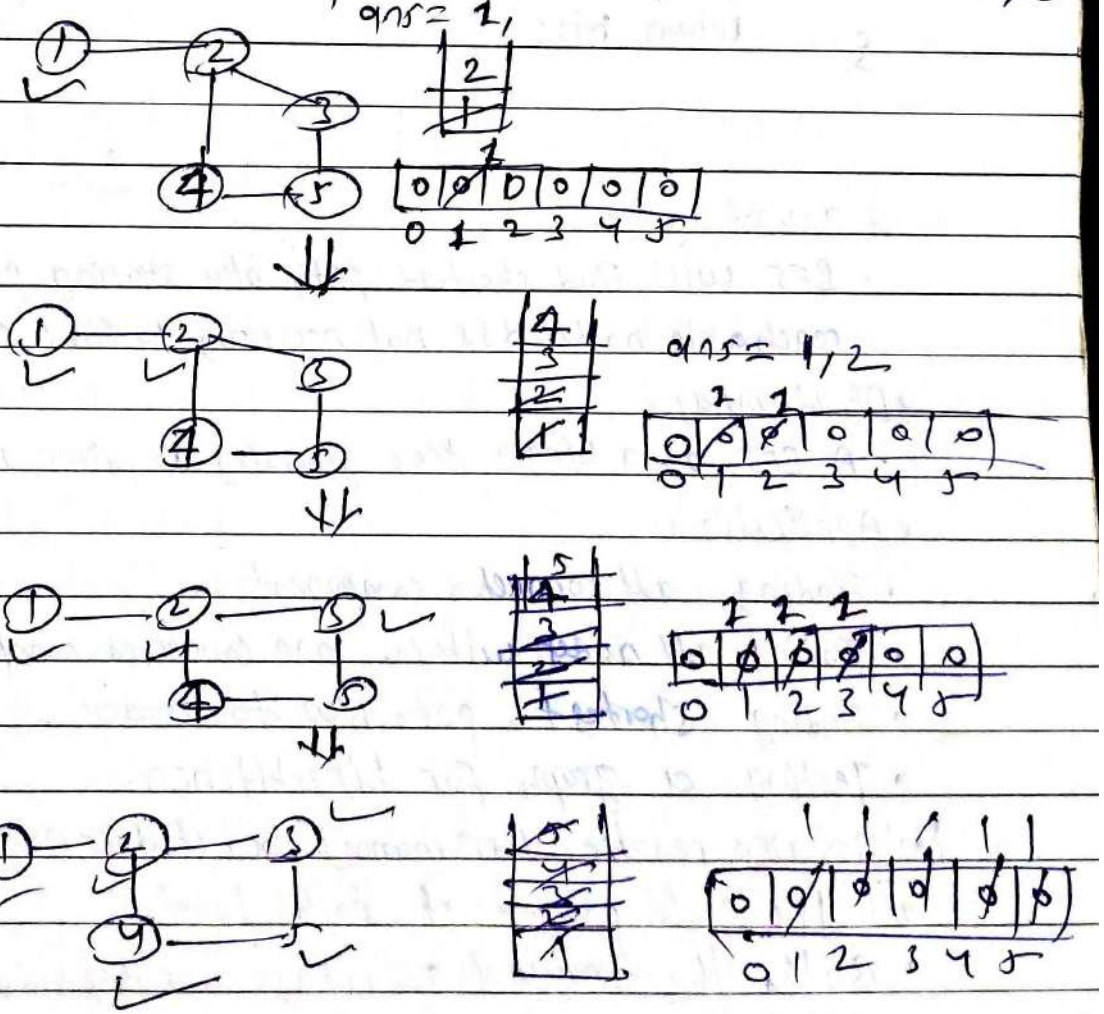
Approach,

① take Queue Data structure, and visited array - an array with all value initialised with 0.

② push 1st node into the queue and mark it visited. after this find its adjacent nodes. which is unvisited node, simply push this adjacent into queue.

③ Now 1st node is node, we will pop out this node from the queue. again process 2nd step until queue is not empty

ex:-



```
vector<int> bfsOfGraph(int V, vector<int> adj[]){
    vector<int> bfs;
    vector<int> vis(V,0);
    queue<int> q;
    q.push(0);
    vis[0]=1;
    while(!q.empty()){
        int node = q.front();
        q.pop();
        bfs.push_back(node);
        for(auto it: adj[node]){
            if(!vis[it]){
                q.push(it);
                vis[it]=1;
            }
        }
    }
    return bfs;
}
```

T.C: O(N+E) | S.C: O(N+E) + O(N) + O(N)
(adj. list, visited, queue.)

* advantages:-
- BFS will find shortest path b/w starting point & any other reachable node. dfs not necessary to find shortest path.

* Disadvantages:-
- A BFS on a binary tree generally requires more memory than DFS.

* Applications:-
- Finding all connected component
- Finding all nodes within one connected component
- Finding shortest path b/w to nodes.
- Testing a graph for bipartiteness.

Note:- DFS require less memory bcz it does not required to store all the child pointer at each level.
- Both the traversal advantage is depending on the data what we are looking for.

(5) Shortest Path Algorithms:-

- shortest path Algorithm is used to solve problem like, to calculate shortest path b/w two point, or we have given a graph and a vertex, we have to find shortest distance b/w given vertex to all vertex of graph.

*Types of shortest path Algorithms:-

(i) Single source shortest path algorithm:- we have given a graph $G=(V,E)$, and a vertex (source) $s \in V$, we have to find shortest path for vertex s to every vertex $v \in V$

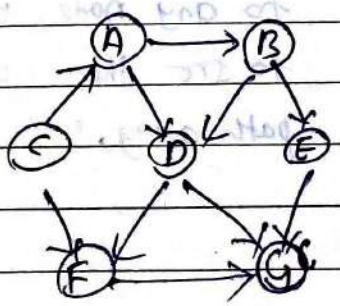
(ii) Single destination shortest path algorithm:- Find the shortest path to a given vertex from every vertex v. By shift the dirn of each edge in the graph, we can shorten this problem to a single-source problem

(iii) Single pair shortest path algorithm:- Find the shortest path from u to v for given vertices u and v.

(iv) All pairs shortest path problem:- Find the shortest path from u to v for every pair of vertices u and v.

(A)* SHORTEST PATH IN UNWEIGHTED GRAPH:—

- Let s be the input vertex. From which we want to find the shortest path to all other vertices. Unweighted graph is a special case of weighted graph with edge weight '1'
- the algorithm is similar to BFS.



let s = s

previous vertex which give. dist[v]

Vertex	Distance [v]	path [v]
A	-1	
B	-1	
C	0	
D	-1	
F	-1	
E	-1	
G	-1	

- '-1' indicating the vertex is examined or not, ∵ it is source that's why make its distance '0'

```
void unweightedShortestPath (vector<int> adj[], int N, int src){
    int dist[N], path[N];
    for (int i=0; i<N; i++) dist[i] = -1;
dist[src]=0; → queue<int> q;
    q.push(src);
    while (q.empty() == false){
        int node = q.front();
        q.pop();
        for (auto it: adj[node]){
            if (dist[it] == -1){
                dist[it] = dist[node] + 1;
                path[it] = node;
                q.push(it);
            }
        }
    }
    // print distance from src to every node
    for (int i=0; i<N; i++) cout<< dist[i] <<" ";
}
```

T.C = O(|E|+|V|), if adjacency list are used.

⇒ Final table

vertex	Dist[i]	path[i]
A	1	C
B	2	A
C	0	-
D	2	A
E	3	B
F	1	C
G	2	F

Note:- "here if we want to print a shortest path from src to any node, or any node to src then we can use path array."

(8) Shortest path in Weighted Graph without negative edge weight [Dijkstra's Algorithm]

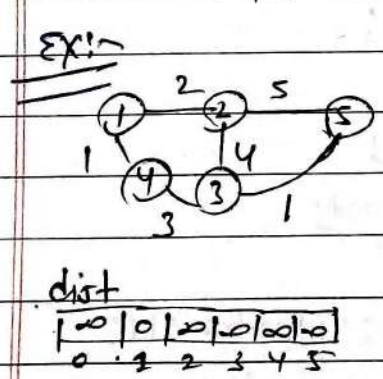
Date
Page No. 311.

- a famous algorithm to find shortest path developed by 'Dijkstra'.
- It is generalization of the BFS algorithm, bcz the regular BFS can not solve the shortest path with given weight the problem is that it cannot guarantee that the vertex at the front of the queue is the vertex closest to source s.
- It does place one constraint on the graph: there can be no negative weight edges.
- It can be used to solve the all-pairs shortest path problem by simply running it on all vertices in V.

Approach:-

(1) we will use a min-heap priority queue and a distance array of size N initialized with infinity (i.e we can not reach there at present) and initialize distance to src node as zero
(2) push source node to priority queue.
(3) For every node at the top of the p.queue pop that and look out for its adjacent nodes. if the distance is less than previous distance we will update the distance and push it in pq.
(4) A node with lower distance would be at the top of the priority queue as opposed to a node with a higher distance. By following the step 3, until pq become empty.

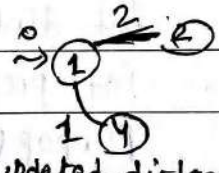
Ex:-



pq
min-heap
(0,1)
<Dist, node>

dist
∞ | 0 | ∞ | ∞ | ∞ | ∞
0 1 2 3 4 5

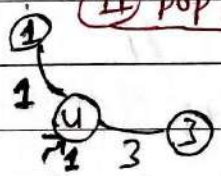
(I) pop (0,1)



(2,4)
(2,2)
(0,1)

updated distance array:
∞ | 0 | 2 | ∞ | 1 | ∞
0 1 2 3 4 5

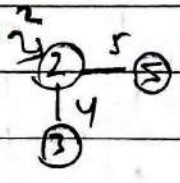
(II) pop (1,4)



(4,3)
(2,4)
(2,2)

∞ | 0 | 2 | 4 | 1 | ∞
0 1 2 3 4 5

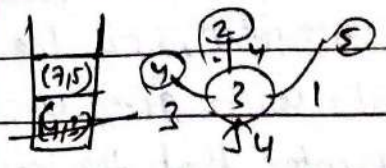
(III) pop (2,



(4,3)
(2,4)
(2,2)

∞ | 0 | 2 | 4 | 1 | 3
0 1 2 3 4 5

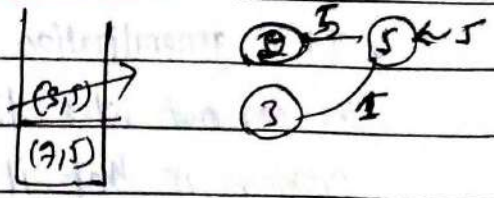
(iv) pop (4,3).



updated distance

∞	0	2	4	1	7
0	1	2	3	4	5

(v) pop (5,5)

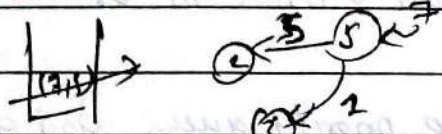


distance will not be update

Note:- here two '5' nodes are present one is with longest distance and other is with lower distance.

⇒ here we can use "set" to store single node with min^m distance

(vi) pop (7,5)



distance will not be update.

∞	0	2	4	1	5
0	1	2	3	4	5

Code:-

```
void Dijkstra(vector<int> adj[], int n, int src){
    priority_queue<pair<int,int>, vector<pair<int,int>>,
                   greater<pair<int,int>>> pq;
    vector<int> dist(n+1, INT_MAX);
distance[src] = 0;
    pq.push(make_pair(0, src));
    while(!pq.empty()){
        int dist = pq.top().first;
        int prev = pq.top().second;
        pq.pop();
        vector<pair<int,int>>::iterator it;
        for(it = adj[prev].begin(); it != adj[prev].end(); it++)
        {
            int next = it->first;      // adjacent node
            int nextDist = it->second; // edge weight
            if(distance[next] > distance[prev] + nextDist){
                distance[next] = distance[prev] + nextDist;
                pq.push(make_pair(distance[next], next));
            }
        }
    }
}
```

```
cout << "The distances from source" << source << "are :\n";
for (int i = 1; i<=n; i++) cout << distance[i] << " ";
return 0;
}
```

TC: $O((N+E)*\log N)$. Going through 'N' nodes & 'E' edges and log N for priority queue.

SC: O(N). Distance array & pq.

Note:- • It uses greedy method: Always pick closes vertex to source.
• uses pq to store unvisited vertices by distance from s.
• does not work with negative weights.

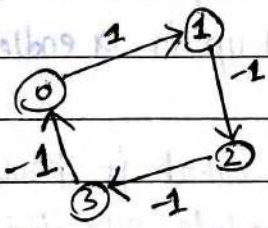
Disadvantages:-
- Blind search, not work with -ve edge weight.

(3) Bellman-Ford Algorithm - Shortest Distance with -ve edge.

• Given a weighted directed graph with -ve edge weights with n nodes and m edges. Nodes are labeled from '0' to 'n-1', the task is to find the shortest distance from the source node to all other nodes. Output "-1" if there exists a negative edge weight cycle in the graph.

Note:- edges[i] is defined as u, v and weight.

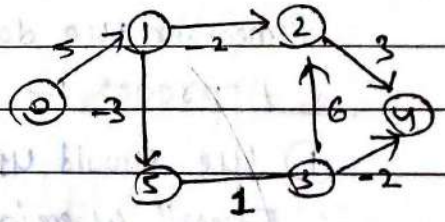
Ex:① Negative edge weight cycle:-



Ans: -1.

• here if we want to go from 0 to 2 then weight will be 1+(-1)=0 but when we again go from (2→3→0→1→2) then distance will be reduced to (-2)
• It can be a negative edge w cycle.

Ex:- ②



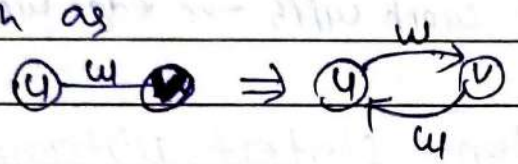
Ans:- distance from source 0 to all node are

0	1	2	3	4	5
0	5	3	-1	-3	-2

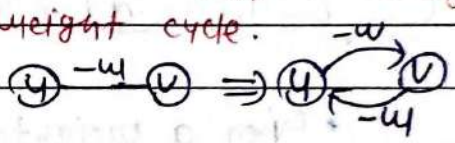
* Why Dijkstra's doesn't work?

- the Dijkstra's algorithm in this might work in some cases and fail in some, due to presence of -ve edge weight.
- in dijkstra's algo, we update the distance array every time we find a better solution which was a lesser distance. but with the presence of -ve edge weights, our algo would continue to update the distance array with lesser & lesser value and we might end up in time limit exceeded or segmentation fault error.

*Note:- it work only one 'directed graph' if we have undirected graph then we can change it into directed graph as



Note:- -ve weight undirected will be result in -ve edge weight cycle.



*Note :- if there are -ve weight cycle. the search for shortest path will go on forever. i.e we can't find shortest path.

* Intution Behind Bellman-Ford Algo :-

we update our distance array in a somewhat same manner as Dijkstra's algo. However we need to handle the -ve edge weight as well. Thus we need to update array in a controlled fashion may be, a specific number of times. Remember, we don't want to end up in a endless loop.

* Approach :-

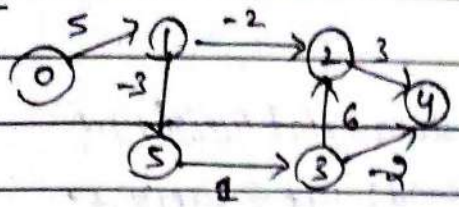
① we would use a technique what is known as "Relaxing Edges" wherein we would update our distance array if we find a better solution. we would do this N-1 times.

② Why 'N-1' times?

= For a given graph with N nodes, the maxm no. of edges we could have b/w any two nodes is 'N-1' in a single

path. Moreover, our adjacency list might be in such a manner that only one node is updated in a single pass.

Ex:-



distance array:-

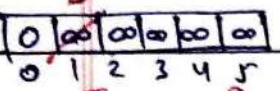
0	∞	∞	∞	∞	∞
0	1	2	3	4	5

→ Relax all edges N-1 times

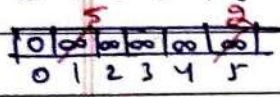
```
if (dist[u] + wt < dist[v])
        dist[v] = dist[u] + wt;
```

Relaxation 1: dist[1] + 6 < dist[2], (∞+6<∞) NO

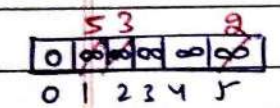
dist[5] + 1 < dist[3], (∞+1<∞) NO



← dist[0] + 5 < dist[1], (0+5<∞) yes (update dist[1] to 5)

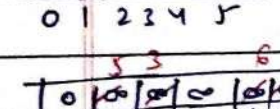


← dist[1] - 3 < dist[5], (5-3<∞) yes (update dist[5] to 2)



← dist[1] - 2 < dist[2], (5-2<∞) yes (update dist[2] to 3)

dist[3] - 2 < dist[4], (∞-2<∞) NO



dist[2] + 3 < dist[4], (3+3<∞) yes (update dist[4] to 6)

updated array after Relaxation 1 is

0	5	3	∞	6	2
0	1	2	3	4	5

(Do N-1=5 times)

Note:- on doing the above Relaxation 4 more times, then our distance array would be as follows:-

0	5	3	-1	-3	-2
0	1	2	3	4	5

(3) Check for negative edge weight cycle :-

- We will now do 1 more Relaxation if there is no updation in distance array then there is no negative edge weight cycle in graph, if there is any updation in distance array means there is negative edge weight cycle.

Note:- So we can use Bellman-Ford to also detect -ve edge weight cycle in graph.

⇒ code :-

```
#include <bits/stdc++.h>
using namespace std;
struct node{
    int u, v, wt;
    node(int first, int second, int weight){
        u = first; v = second; wt = weight;
    }
};
int main(){
    int N = 6, m = 7
    vector<node> edges;
    edges.push_back(node(0,1,5));
    edges.push_back(node(1,2,-2));
    edges.push_back(node(1,5,-3));
    edges.push_back(node(2,4,3));
    edges.push_back(node(3,2,6));
    edges.push_back(node(3,4,-2));
    edges.push_back(node(5,3,1));
    int src=0;
    int inf = 10000000;
    vector<int> dist(N, inf);
    dist[src] = 0;
```

Relaxation "N-1" times

```
    for(int i=1; i<=N-1; i++){
        for(auto it: edges){
            if(dist[it.u] + it.wt < dist[it.v]){
                dist[it.v] = dist[it.u] + it.wt;
            }
        }
    }
    int fl=0;
```

checking for -ve edges cycle

```
    for(auto it: edges){
        if(dist[it.u] + it.wt < dist[it.v]){
            cout<<-1;
            fl=1;
            break;
        }
    }
```

```
if(FP==0){
    for(int i=0; i<N; i++){
        cout<<dist[i]<<" ";
    }
    return 0;
}
```

O/P:- 0 5 3 3 1 2

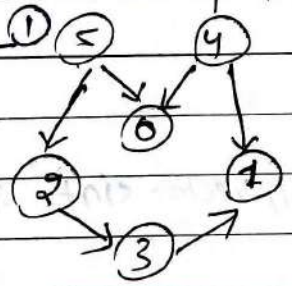
TC: O(N*E)

SC: O(N)

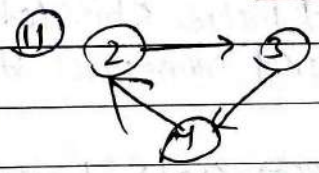
(6) Topological Sort (Kahn's Algorithm) :-

- Assume that we need to shedule a series of tasks, where we cannot start one task until after its prerequisites are completed. we wish to organize the tasks into a linear order that allow us to complete them one at a time without violating any prerequisites.
- This type of problem can be model using DAG. the graph is directed bcz one task is prerequisite of another. it is acyclic bcz a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite.
- The process of laying out the vertices of DAG in a linear order to meet prerequisite rules is called a 'topological sort'.

Ex:



O/P:- 4 5 2 0 3 1
or, 4 5 0 2 3 1
or, 5 4 2 0 3 1
or, 5 4 0 2 3 1



2 → 3 → 4 → 2

Topological sort not possible here, bcz it should be DAG. here every node has prerequisites.

* Application :-

- Representing course prerequisites
- Detecting deadlock
- pipline of computing jobs
- Checking for symbolic link loop
- Evaluating formula in spread sheet.

*Implementation:-

- First we have to start from a node that doesn't have any previous edges (prerequisites). to find it we use the concept of in-degree (no. of edge pointing toward a node). we use in-degree concept to find topological sorting as following:-

(1) In order to maintain in-degree of each node we take an array of size V (no. of nodes). Find in-degree and fill them in array.

(2) Add all node in a queue having in-degree '0'. and simply apply bfs on queue with some condition.

→(i) take top from queue (pop them). add this to our answer

→(ii) Now take neighbours of poped node and reduce it in-degree by 1 (bcz one prerequisite have processed).

→(iii) after reducing in-degree by 1. if the indegree become zero add it to the queue.

(3) apply the above 3 condition with BFS till queue become empty.

Note:- if queue become empty without printing all nodes then graph contains a cycle.

*code:-

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

vector<int> topological (int N, vector<int> adj[]){
    queue<int> q;
    vector<int> indegree (N,0);
    for(int i=0; i<N; i++){
        for(auto it: adj[i]){
            indegree[it]++;
        }
    }
    for(int i=0; i<N; i++){
        if(indegree[i]==0){
            q.push(i);
        }
    }
```

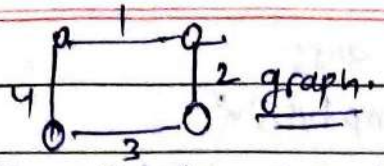
```
vector<int> ans;
while (!q.empty()){
    int node = q.front();
    q.pop();
    ans.push_back(node);
    for(auto it : adj[node]){
        indegree[it]--;
        if(indegree[it]==0){
            q.push(it);
        }
    }
}
return ans;
}

int main(){
    vector<int> adj[6];
    adj[5].push_back(2);
    adj[5].push_back(0);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[3].push_back(1);
    adj[2].push_back(3);
    vector<int> v = topological(6, adj);
    for(auto it:v) cout<<it<<" ";
    return 0;
}
```

O/P:- 4 5 2 0 3 1

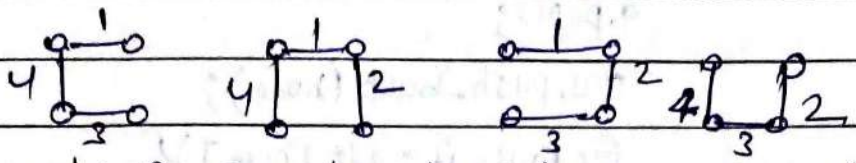
TC: O(N+E)
SC: O(N)+O(N)

(7) Minimal Spanning Tree.

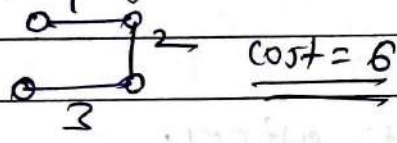
- The spanning tree of a graph is a subgraph that contains all the vertices and it also a tree. Total edge (n-1) with 'n' nodes.
- A graph may have many spanning Tree.

Ex:-  graph.

- spanning Trees for the given graph can be:-



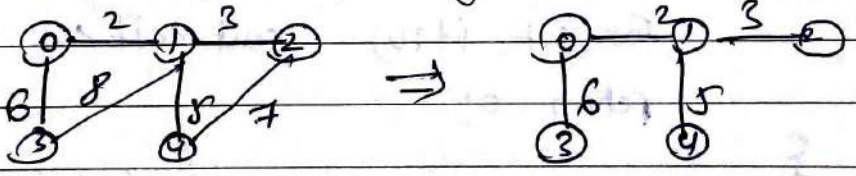
- The cost of spanning tree is sum of weights of all the edges in tree
- minimum spannig tree (MST) is the spanning tree when the cost is minimum among all spanning tree. there also can be many minimum spaning tree.
- Minimum spanning tree for given graph will be



- A minimum spanning tree exist only if graph is connected.
- There are two famous algorithm to find MST.
  (i) Paim's Algorithm (ii) Krustal's Algorithm.

(1) Paim's Algorithm:→

Given a weighted, undirected and connected graph of V vertices & E edges. The task is to find the sum of weights of the edges of the minimum spanning tree.

Ex:- 

Intution:- Let's start with anyone node in our graph. As the 1st step, we find out all the adjacent edges connected to this node and then pick up the minimum one. Now we have 2 nodes. we further continue this process but now we would consider all the edges connected to these two nodes and pick up the minimum one. we then continue this process until all the nodes are covered.

* we would have 3 arrays (key, mst & parent)
- key: this array holds the weight/cost of MST (initialize to INT_MAX except index 0 which is set with value of zero)
- mst: boolean array which indicates whether a node is already a part of MST or not (initialize to false except 0)
- Parent: indicates parent of a particular node in MST (initial -1)

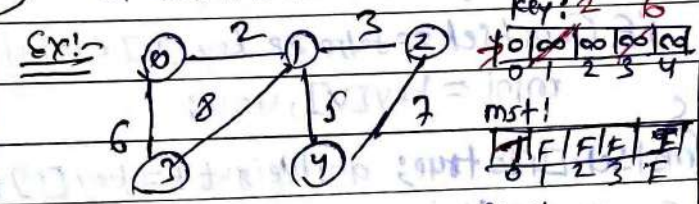
Step:—

Note:- when we picking the node with minimum weight (from key) to include in mst mark it as visited in mst array and also set it's parent in parent array (parent is that node whose adjacent is this)

(key will be update only when weight is min from previous store key-value)

- we will pick minm weight node from key and also we will update key array of their adjacent node weight.
- we will continue this process untill all nodes become part of the MST.

Ex:-



key: ∞ ∞ ∞ ∞ ∞
0 1 2 3 4

mst: F F F F F
0 1 2 3 4

Start:- minm=0, update mst

update parent

Parent: -1 0 0 0 0
0 1 2 3 4

① Take minm from key
update all adjacent in key.

key: ∞ 2 3 6 5
0 1 2 3 4

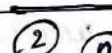
*Pick '1'

mst: T T F F F
0 1 2 3 4

mark '1' in mst

Parent: -1 0 1 0 1
0 1 2 3 4

Update parent of '1'



key: 0 2 3 6 5
0 1 2 3 4

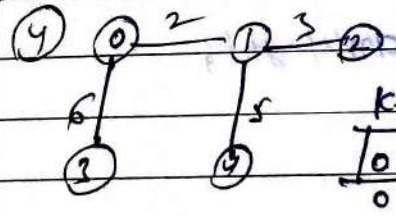
mst: T T T F F
0 1 2 3 4

Parent: -1 0 1 0 1
0 1 2 3 4

key: 0 2 3 6 5
0 1 2 3 4

mst: T T T F T
0 1 2 3 4

Parent: -1 0 1 0 1
0 1 2 3 4



key: 0 2 3 6 5
0 1 2 3 4

mst: T T T T T
0 1 2 3 4

parent: -1 0 1 0 1
0 1 2 3 4

- now all nodes are visited so we stop this process
- we can run loop (N-1) time to visit all node (i.e pick N-1 time minm from key).

code:- (Brute force):-

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

void printmst( int N, vector<pair<int,int>> adj[]){
    int parent[N];
    int key[N];
    bool mstSet[N];
    for(int i=0; i<N; i++)
        key[i]=INT_MAX, mstSet[i]=false;
    key[0]=0; parent[0]=-1;
    int ansWeight=0;
    for(int count=0; count<N-1; count++){
        int mini=INT_MAX, u;
        for(int v=0; v<N; v++){
            if(mstSet==false && key[v]<mini)
                mini=key[v], u=v;
        }
        mstSet[u]=true; ansWeight+=key[u];
        for(auto it: adj[u]){
            int v=it.first;
            int weight=it.second;
            if(mstSet[v]==false && weight<key[v])
                parent[v]=u, key[v]=weight;
        }
    }
    for(int i=1; i<N; i++)
        cout<<parent[i]<<" - "<<i<<"\n";
    cout<<"weight is"<<ansWeight;
}

int main(){
    int N=5, m=6;
    vector<pair<int,int>> adj[N];
    for(int i=0; i<m; i++){
        int u; int v; int wt;
```

```
    cin >> u >> v >> wt;
    adj[u].push_back({v, wt});
  }
  printmst(N, adj);
  return 0;
}
```

Time C: O(N^2)
SC: O(N)

Code :- (optimized) :-

- In Brute force we are going through the key-value again and again in whole array to find minm edge weight.
- we would use a minm heap to facilitate this task that stores stores the minm weight at the top (priority queue)
- min-heap would contain the weight & node.

Note:- in brute force code only red block will be change black block code will be same.

```
priority_queue<pair<int,int>, vector<pair<int,int>>, greater
                                    <pair<int,int>>> pq;
pq.push({0,0});
while(!pq.empty()){
    int u = pq.top().second;
    pq.pop();
    mstSet[u] = true; ansWeight += key[u];
    for(auto it: adj[u]){
        int v = it.first;
        int weight = it.second;
        if(mstSet[v] == false && weight < key[v]){
            parent[v] = u;
            key[v] = weight;
            pq.push({key[u], v});
        }
    }
}
```

TC: O(NlogN), N iteration and logN for priority queue.
SC: O(N), array and priority queue

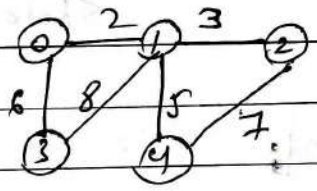
② Kruskal's Algorithm :-

Given a weighted, undirected, and connected graph of V vertices & E edges. the task is to find sum of weights of the edges of MST

Note:- . It uses Disjoint set data structure.
- → FindParent() or, FindSet() :- to find parent of node
- → union() or, unionSet() :- to join two component.

Algorithm :-
- sort all edges according to their weight.
- Greedily pick minimum edge and make sure two nodes belong to different components (using disjoint set ds FindParent operation). if they belong to same component it would indicate, we would be having cycles which is not possible in a MST.
- Also, once a node be a part of the MST, we must join the two component using the union operation of DSU.

Ex:-

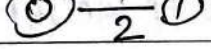


wt	u	v
2	0	1
6	0	3
2	1	0
8	1	3
5	1	4
3	1	2
6	3	0
8	3	1
5	4	1
7	4	2

⟹ Sorting

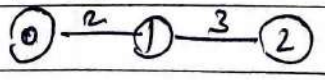
wt	u	v
✓ 2	0	1
✓ 2	1	0
✓ 3	1	2
✓ 5	1	4
✓ 5	4	1
✓ 6	0	3
× 6	3	0
× 7	4	2
× 8	1	3
× 8	3	1

→ Node ⓪ and ① (✓) belong to different component. Hence, add in mst

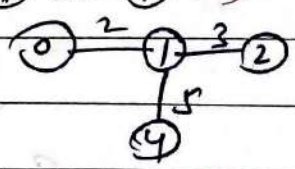


→ the second entry ② and ⓪ (✗) not consider it shared common parent.

→ ① and ② (✓)

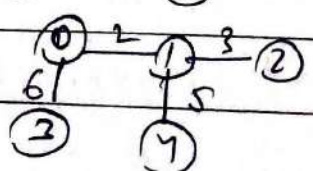


→ ① and ④ (✓)



→ ④ and ① (✗)

→ ⓪ and ③



code:-

```
#include<bits/stdc++.h>
using namespace std;
struct node{
    int u, v, wt;
    node(int first, int second, int weight){
        u = first;
        v = second;
        wt = weight;
    }
};
bool comp(node a, node b){
    return a.wt < b.wt;
}
int findPar(int u, vector<int> &parent){
    if(u == parent[u]) return u;
    return parent[u] = findPar(parent[u], parent);
}
void unionn(int u, int v, vector<int> &parent, vector<int> &rank){
    u = findPar(u, parent);
    v = findPar(v, parent);
    if(rank[u] < rank[v]) parent[u] = v;
    else if(rank[v] < rank[u]) parent[v] = u;
    else{ parent[v] = u;
        rank[u]++;
    }
}
int main(){
    int N=5, M=10;
    vector<node> edges;
    for(int i=0; i<M; i++){
        int u, v, wt;
        cin>>u>>v>>wt;
        edges.push_back(node(u, v, wt));
    }
```

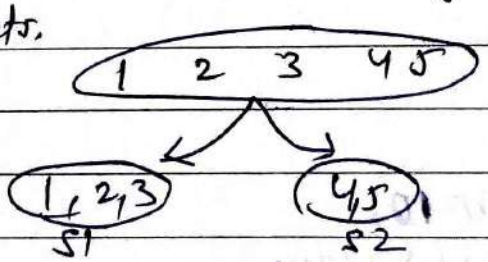
```
sort(edges.begin(), edges.end(), comp);
vector<int> parent(N);
for(int i=0; i<N; i++) parent[i]=i;
vector<int> rank(N,0);
int cost=0;
vector<pair<int,int>> mst;
for(auto it:edges){
    if(findPar(it.v, parent) != findPar(it.u, parent)){
        cost+=it.wt;
        mst.push_back({it.u,it.v});
        unionn(it.u, it.v, parent, rank);
}  }
cout<<cost<<endl;
for(auto it:mst) cout<<it.first<<" - "<<it.second<<endl;
return 0;
}
```

o/p
1 6
0 - 1
1 - 2
1 - 4
0 - 3

TC: O(ElogE) + O(E*4*alpha) ≈ O(ElogE) + O(E) (constant)
ElogE for sorting and E*4*α for find parent operation 'E' times.
SC: O(N). Parent array + Rank array

(8) Disjoint Set (union by rank, Find, path compression):-

- Two or more sets with nothing in common are called disjoint sets.



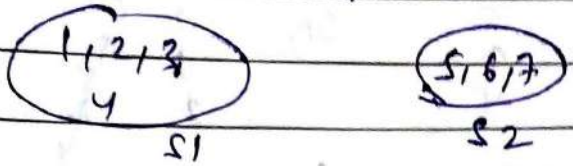
$S1 \cap S2 = \phi$
(Disjoint set)

* Use of DS:-

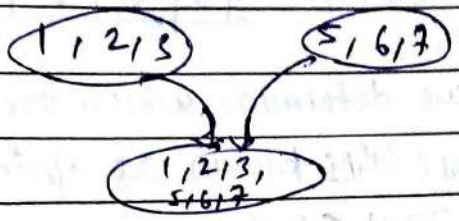
(1) keeps track of the set that an element belong to: It is easier to check, given 2 element, whether they belong

to the same subset. (known as find operation)

S(1) == S(2) ?



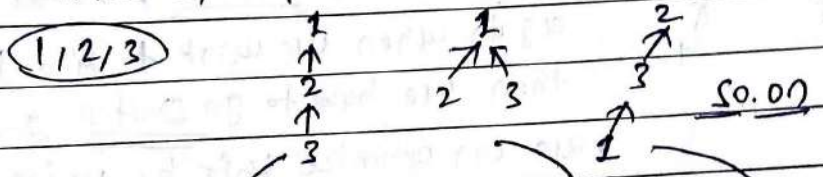
(2) Used to merge two set into one. (UNION operation)



• Union of 2 element is same as union of 2 SETS

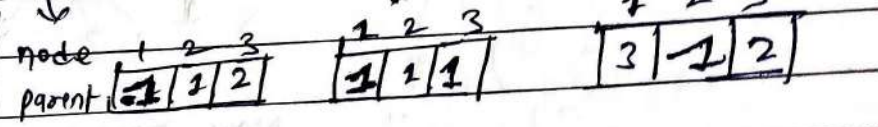
* Implementation :-

• Disjoint set uses chaining to define a set. The chaining is defined by a parent-child relationship.

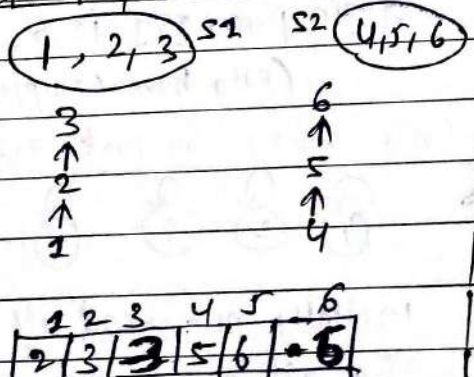


Note:- Structure does not matter.

Representation using parent array



• 2 sets Representation:



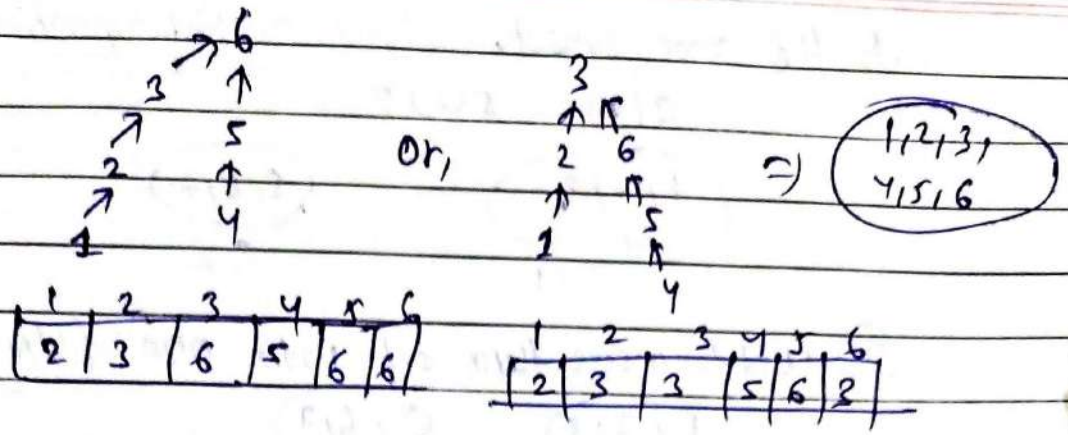
• Find and union of 2 set

Q. Are (2,5) in same set? (Find)

parent[2] is 3 and parent[5] is 6 hence it is not in same set

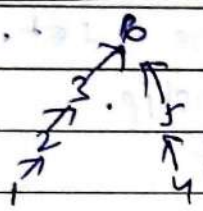
Q. if not in same set then merge them. (Union)

• To merge two set or two element (2,5) we can make parent of 3 to 6 or parent of 6 to 3. i.e.

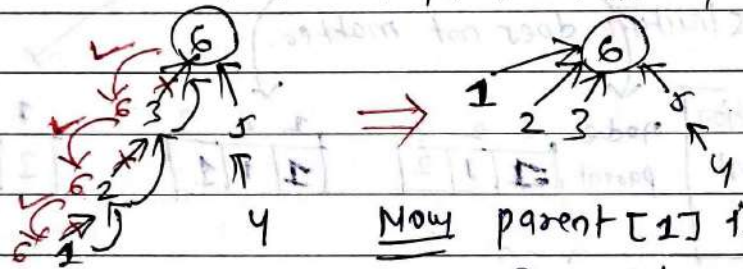


Note:- How can we determine which one is optimized. For this we use technique known as union by rank here, rank of 3 and 6 both are same so we can use anyone.

* Path Compression :—



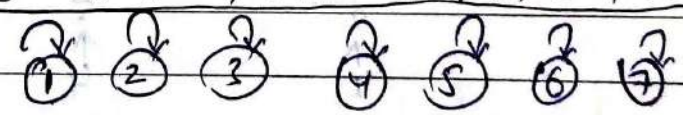
- here if we want to find parent of '1' then we have to go 3-step 1→2→3→6, and again when we want to find parent of '2' then we have to go 2-step 2→3→6,
- we can optimize this by using path compression



Now parent[1] is 6, 3-step
(b<2 path will compress now)
again paren[2] is 6, 1-step.
(Path have compressed)

(*) Example :— (use union by rank and path compression)

union (1,2)
union (1,3)
union (4,5)
union (6,7)
union (5,6)
union (3,7)



initially parent of all node is itself i.e all nodes are set itself.

∴ parent array will be

	1	2	3	4	5	6	7
parent	1	2	3	4	5	6	7

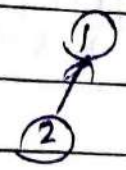
and rank array will be zero for all

	1	2	3	4	5	6	7
rank	0	0	0	0	0	0	0

① union (1,2)

• findPar(1) = 1 i.e
• findPar(2) = 2

i.e both are two set



parent:

1	1	3	4	5	6	7
1	2	3	4	5	6	7

• rank[2] is 0.
• rank[2] is 0

rank:

1	0	0	0	0	0	0
1	2	3	4	5	6	7

since rank is same and we are making parent of 2 as 1 so increase rank of 1 by 1.

② union (1,3)

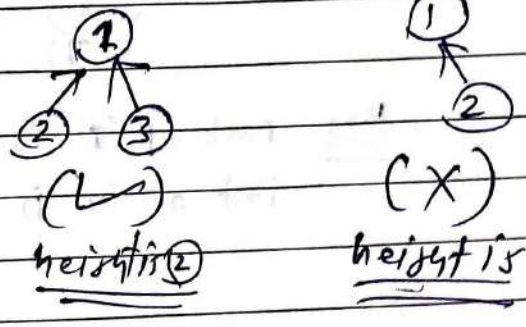
• parent[1] = 1
• parent[3] = 3

i.e it is in two different set

• rank[1] is 1.
• rank[3] is 0

here rank[1] > rank[3]

∴ make parent of 3 as '1'



Note:- • we are making rank higher as parent bcz if we will make as this height of tree will be same (minm)

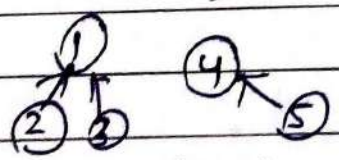
• and in this case rank will not be change rank will be increase only when the two parent rank will be same

Q. why we making minm height?

bcz when we will find parent will take minm step.

May parent | 1 | 1 | 1 | 4 | 5 | 6 | 7 |, rank | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
1 2 3 4 5 6 7 1 2 3 4 5 6 7

③ union (4,5)



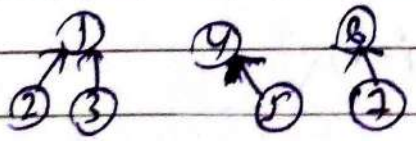
parent

2	2	1	4	4	6	7
1	2	3	4	5	6	7

, rank

1	0	0	1	0	0	0
1	2	3	4	5	6	7

④ union (6,7)



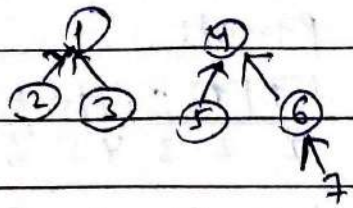
parent

2	1	1	4	4	6	6
1	2	3	4	5	6	7

rank

2	0	0	1	0	1	0
1	2	3	4	5	6	7

⑤ union (5,6)



parent

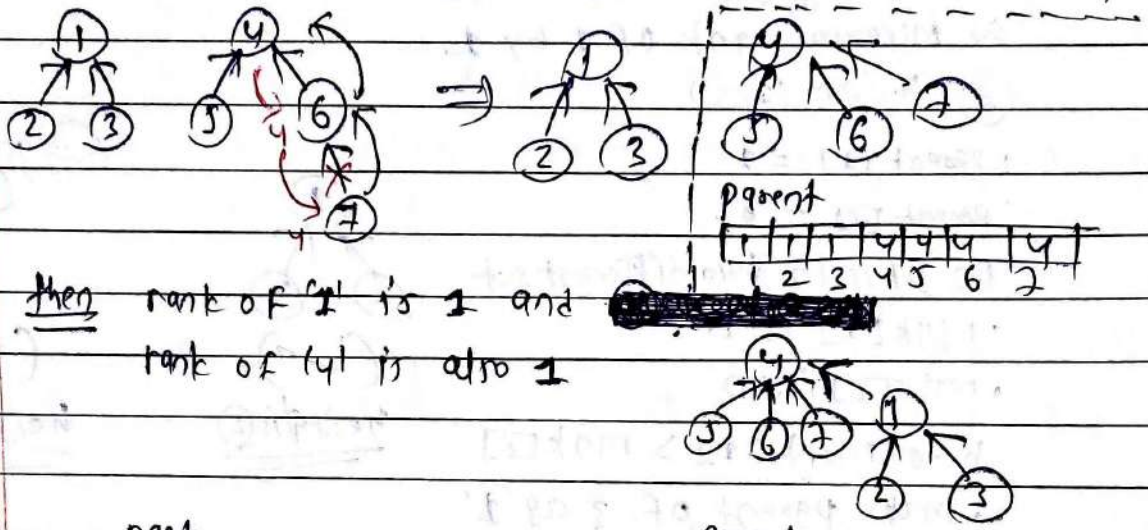
1	2	3	4	5	6	7
1	1	1	4	4	4	6

rank

1	2	3	4	5	6	7
1	0	0	2	0	1	0

⑥ union (3,7)

Note:- here when we are going to find parent of '7' at that time we will do path compression.



parent

1	2	3	4	5	6	7
1	1	1	4	4	4	4

Then rank of '4' is 1 and rank of '4' is also 1

rank

1	2	3	4	5	6	7
1	0	0	2	0	1	0

parent

1	2	3	4	5	6	7
4	1	1	4	4	4	4

Q Why there is use of rank concept not height?
bcz at the time of path compression height will be change but we can keep rank same was before.

Code:—

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

int parent[100000];
int rank[100000];

void makeSet(){
    for(int i=1; i<=n; i++){
        parent[i]=i;
        rank[i]=0;
    }
}
```

```
int findPar(int node){
    if (node == parent[node]){
        return node;
    }
    return parent[node] = findPar(parent[node]);
}

void union(int u, int v){
    u = findPar(u);
    v = findPar(v);
    if (rank[u] < rank[v]){
        parent[u] = v;
    }else if (rank[v] < rank[u]){
        parent[v] = u;
    } else{
        parent[v] = u;
        rank[u]++;
    }
}

void main(){
    makeSet();
    int m;
    cin>>m;
    while(m--){
        int u,v; ---> cin>>u>>v;
        union(u,v);
    }
//if 2 and 3 belong to the same component or not
    if(findPar(2) != findPar(3)){
        cout<<"Different Component";
    }else{
        cout<<"Same component";
    }
}
```

TC: O(E*4)
SC: O(E)

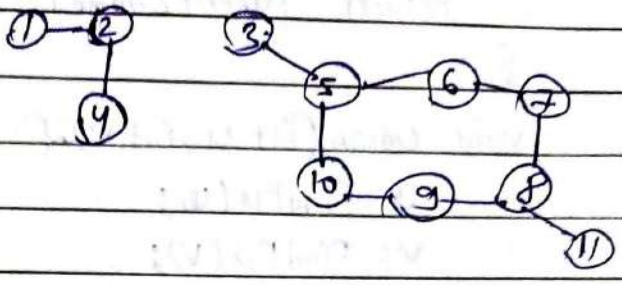
(9) Detect a cycle in undirected graph using BFS

Ex:- Input :
Adjacent list:.

1 - 2
2 - 1 4
3 - 5
4 - 2
5 - 3 10 6
6 - 5 7
7 - 6 8
8 - 7 9 11
9 - 10 8
10 - 5 9
11 - 8

Output:- Yes

Explanation



Intuition:- check for the visited element if it is found again, this means the the cycle is present in the given undirected graph.

Note:- Here in queue we will insert (node, prev), for example if we are inserting 2 adjacent then (1,2) and (4,2) will be insert in queue. bcz after this when we are going to insert 4 adjacent then (2) is visited already but it does not making a cycle so we will check if the previous/parent of 4 is same as adjacent 2, then this will not create cycle, here prev of 4 was 2 which is same. hence it will not create a cycle.

- In case of edges 7 - 8 and 9 - 8 the previous/parent of 8 will be different hence these are making cycle.
- Hence by inserting (node, prev) we are making sure that '5 - 10' and '10 - 5' will not create cycle.

ex:-

code :-

```
class Solution {
public:
    bool checkForCycle(int s, int V, vector<int> adj[],
                       vector<int> &visited) {
```

```
queue<pair<int,int>> q;
visited[s] = true;
q.push({s, -1});
while(!q.empty()){
    int node = q.front().first;
    int par = q.front().second;
    q.pop();
    for(auto it: adj[node]){
        if(!visited[it]){
            visited[it]=true;
            q.push({it, node});
        }else if(par != it) return true;
    }
}
return false;
}
bool isCycle(int V, vector<int> adj[]){
    vector<int> vis(V+1, 0);
    for(int i=1; i<=V; i++){
        if(!vis[i])
        {
            if(checkForCycle(i,V,adj,vis)) return true;
        }
    }
};
```

T.C: O(N+E), S.C: O(N+E) + O(N) + O(N)
adjacent list, array, queue.

(10.) Detect a cycle in undirected graph using DFS

Same concept will be use as in BFS, but here we will do DFS traversal instead of BFS.

T.C: O(N+E)

S.C: O(N+E) + O(N) + O(N)

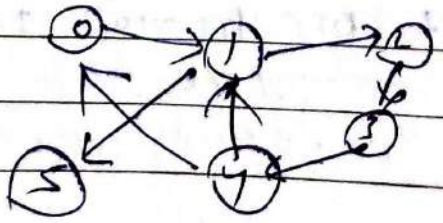
```
class Solution{
  public:
    bool checkforcycle(int node, int parent, vector<int>&vis,
                                    vector<int> adj[])
    {
        vis[node] = 1;
        for(auto it: adj[node]){
            if(!vis[it]){
                if(checkforcycle(it, node, vis, adj))
                    return true;
            }else if(it != parent){
                return true
            }
        }
        return false;
    }
  public:
    bool isCycle(int V, vector<int> adj[]){
        vector<int> vis(V+1, 0);
        for(int i=0; i<V; i++){
            if(!vis[i]){
                if(checkforcycle(i,-1, vis,adj))
                    return true;
            }
        }
        return false;
    }
};
```

(11.) Detect A Cycle in Directed Graph using DFS.

- Given a 2D adjacency list representation of a directed graph. Check whether the graph has cycle or not.

Input:- 0 → 1
1 → 2, 5
2 → 3
3 → 4
4 → 0, 1
5 → null

Output:- yes



Note:- we can not use here undirected graph cycle detection approach bcz ok to that for graph like ⑤→⑥ will give cycle bcz ⑥ parent are ⑤ & ⑦ which is not equal

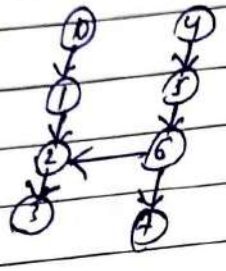
but it is not cycle

Date
Page No. 33

Intuition:- The basic intuition for a cycle detection is to check whether a node (when we are processing it) is reachable or not with its neighbours and also its neighbours' neighbours.

- For this, along with visited array we will use a extra array which indicate whether we are processing a node's neighbour or not. if we are processing a node neighbours and again we have reached at that nodes, means there is cycle. ex:- 0 → 1 → 2 → 3 → 4 → 0

Note:- if there is a graph like this :-



- here, if we will not use an extra array, and we will check only with visited array the it will show cycle.
- but by using extra array it will not show a cycle.

bcz in extra array we can store information that we are not processing ② node's neighbours

⇒ Code:-

```
class Solution {
    private:
    bool checkCycle (int node, vector<int> adj[], int vis[],
                                                int dfsVis[]) {
        vis[node] = 1;
        dfs[node] = 1;
        for (auto it: adj[node]) {
            if (!vis[it]) {
                if (checkCycle(it, adj, vis, dfsVis)) return true;
            } else if (dfsVis[it]) {
                return true;
            }
        } // loop end
        dfsVis[node] = 0; return false;
    } // function end
```

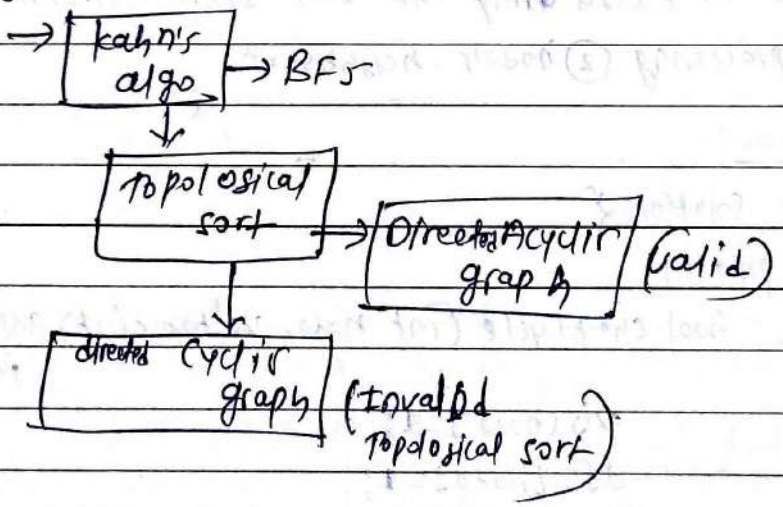
if we have visit this node then check we are currently visit that neighbour and neighbour's neighbours

```
public:
    bool isCyclic(int N, vector<int> adj[]){
        int vis[N], dfsVis[N];
        memset(vis, 0, sizeof vis);
        memset(dfsVis, 0, sizeof dfsVis);
        for(int i=0; i<N; i++){
            if(!vis[i]){
                if(checkCycle(i, adj, vis, dfsVis)){
                    return true;
                }
            }
        }
        return false;
    }
};
```

TC: O(V+E) / SC: O(V)

(12) Cycle Detection in Directed Graph (kahn algo) using BFS

- using topological sort (kahn algo):-



Note:- in topological sort if we are not able to cover all the node it means there is a cycle (we can not cover a node only when their in-degree will not be zero in any way). so we will count the node after popping from queue and at the end check if it is equal to no. of total node or not.

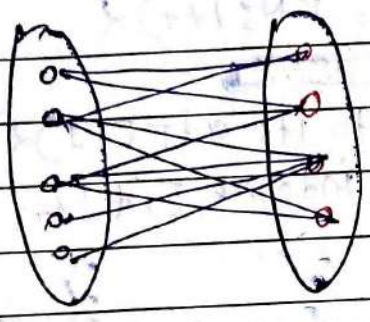
Code:-

```
Class Solution{
public:
    bool isCyclic(int N, vector<int> adj[]){
        queue<int> q;
        vector<int> indegree(N,0);
        for(int i=0; i<N; i++){
            for(auto it: adj[i]){
                indegree[it]++;
            }
        }
        For(int i=0; i<N; i++){
            if(indegree[i]==0){
                q.push(i);
            }
        }
        int cnt=0;
        while(!q.empty()){
            int node=q.front();
            q.pop();
            cnt++;
            for(auto it: adj[node]){
                indegree[it]--;
                if(indegree[it]==0){
                    q.push(it);
                }
            }
        }
        if(cnt==N) return false;
        return true;
    }
};
```

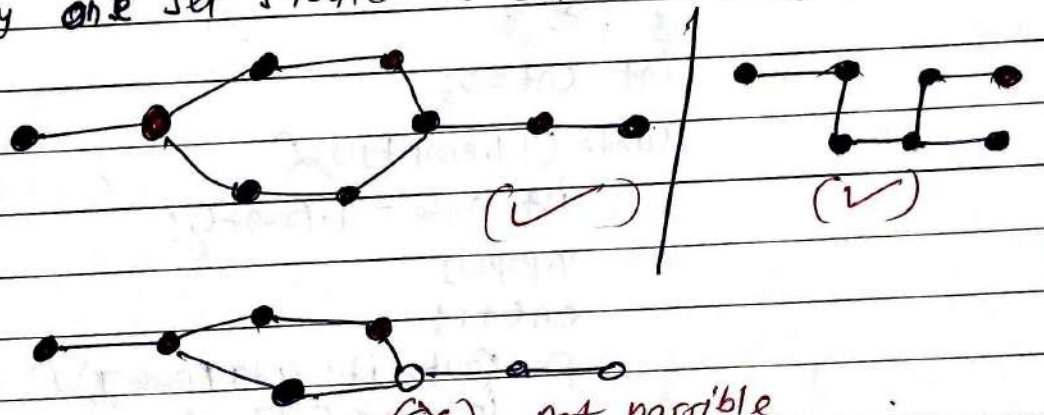
T.C: (V+E)
S.C: O(V)

(13) Bipartite Graph using BFS (Graph coloring)

- A Bipartite graph is a graph whose vertices can be divided into two independent sets, U and V. such that every edge (u,v) either connects a vertex from U to V or a vertex from V to U. i.e either u belongs to U and v belongs to V or $v \in U$ and $u \in V$. but (u,v) can not belong to same set



- A bipartite graph is possible if the graph coloring is possible using two color such that no color ar ajacent, or we can say one set should be color with only one color.



(X) not possible

Note:- here we can notice if in a graph contain cycle with odd no. of nodes then it is not a bipartite graph, otherwise it is bipartite

- we can solve this problem also using backtracking algorithm in coloring problem.

* Approach By using BFS :-

- take 2 color as 0 and 1 First color all node with -1 by using array.
- take node and color with 0 or, 1 again using BFS, update in array.

src

color it's all adjacent node with node's opposite color.

- before coloring adjacent node check if it colored before or not, if it is colored, then check it is opposite color or not, if adjacent colored is not colored with opposite color, means it is not bipartite. return false.
- if it is colored, then repeat above steps until queue become empty with opposite with BFS, then return true.

Code:-

```
class Solution {
    private:
    bool bipartiteBfs (int src, vector<int> adj[], int color[])
    {   queue<int> q;
        q.push (src);
        color[src]=1;
        while (!q.empty ()){
            int node=q.front();
            q.pop();
            for (auto it: adj[node]){
                if (color[it] ==-1){
                    color[it] = 1-color[node];
                    q.push (it);
                }else if (color[it] == color[node]){
                    return false;
                }
            }
        }
        return true;
    }
    bool checkBipartite (vector<int> adj[], int n){
        int color[n];
        memset (color, -1, sizeof color);
        for(int i=0; i<n; i++){
            if (color[i]==-1){
                if(!bipartiteBfs(i, adj, color)){
                    return false;
                }
            }
        }
        return true
    }
};
```

TC: O(V+E)
SC: O(V+E) + O(V) + O(V)

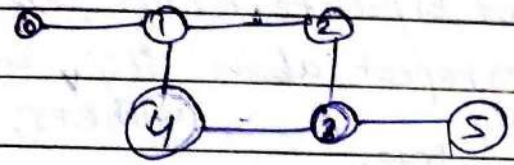
Date

Page No. 340.

(14) Bipartite graph check using DFS:-

We will follow same approach done in previous using BFS. but the node traversing will be in DFS manner.

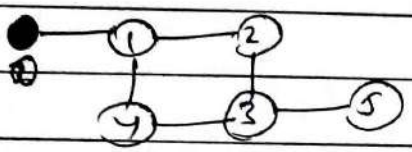
Ex:- initial color = -1 for all node



color

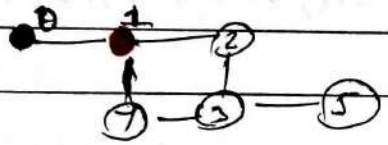
-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

(I)



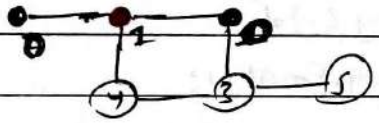
0	-1	-1	-1	-1	-1
0	1	2	3	4	5

(II)



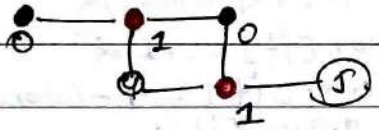
0	1	-1	-1	-1	-1
0	1	2	3	4	5

(III)



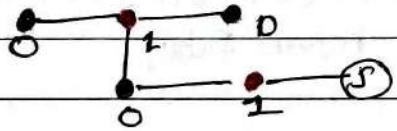
0	1	-1	-1	-1	-1
0	1	2	3	4	5

(IV)



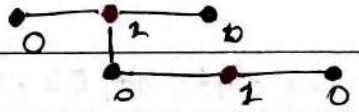
0	1	0	-1	-1	-1
0	1	2	3	4	5

(V)



0	1	0	1	-1	-1
0	1	2	3	4	5

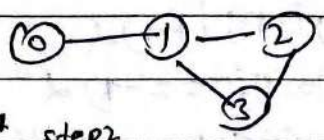
(VI)

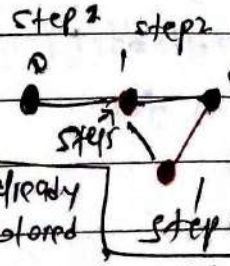


0	1	0	1	0	-1
0	1	2	3	4	5

hence it is bipartite

Ex:- (2)





0	1	0	1
0	1	2	3

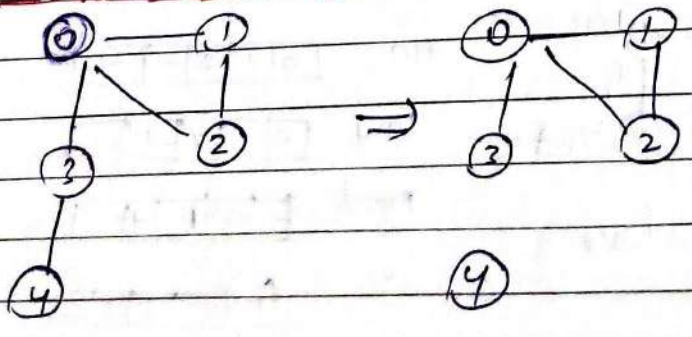
∴ not bipartitite.

Code :-

```
class Solution{
    bool bipartiteDfs (int node, vector<int> adj[], int color[]
    {
        for(auto it: adj[node]){
            if(color[it]==-1){
                color[it] = 1-color[node];
                if (!bipartiteDfs (it, adj, color)){
                {      return false;
                }
            else if (color[it]==color[node]) return false;
        }
        return true;
    }
    bool checkBipartite (vector<int> adj[], int n){
        int color[n];
        memset (color, -1, sizeof color);
        for(int i=0; i<n; i++){
            if (color[i]==-1){
                color[i]=1;
                if (!bipartiteDfs (i, adj, color)){
                    return false;
                }
            }
        }
        return true;
    }
};
```

TC: O(V+E) | SC: O(V)

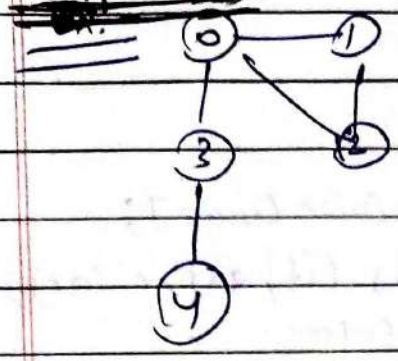
(15) Bridges in a graph :-



after removing 3-4 edge it become two disconnected graph hence it is bridge of graph.

- The bridges of a graph are edges by removing those graph will be disconnected into two or more parts.

Approach :-



tin

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

low

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

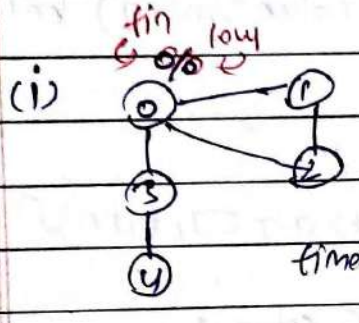
parent

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

vis

F	F	F	F	F
0	1	2	3	4

(i)



timer = 0/1

tin

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

low

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

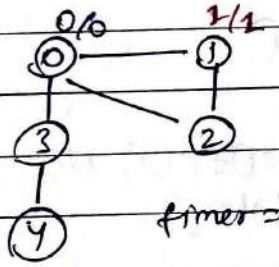
parent

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

vis

T	F	F	F	F
0	1	2	3	4

(ii)



timer = 1/2

tin

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

low

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

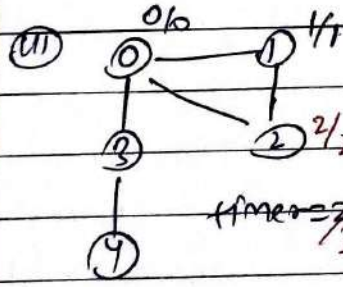
parent

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

vis

T	T	F	F	F
0	1	2	3	4

(iii)



timer = 2/3

tin

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

low

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

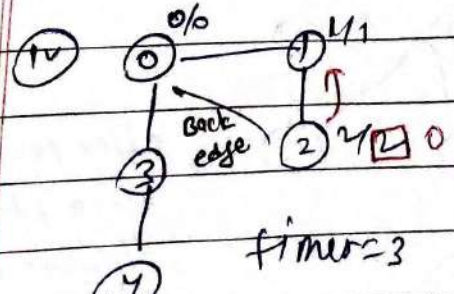
parent

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

vis

T	T	T	F	F
0	1	2	3	4

(iv)



timer = 3

tin

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

low

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

parent

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

vis

T	T	T	F	F
0	1	2	3	4

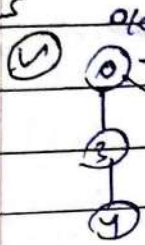
if (neighbour == parent) continue;
else {
now in this case there is adjacent '0' of '2' which is not parent of '2' & visited it means we could reach from '0' to '2' also. so we check lowest possible time to reach '2', if from '0' we can reach earliest then update the lowest time of '2'

low[node] = min [low[node], tin[neighbour]]

↳ this will done only if neighbour is not parent of node and also neighbour should be visited.

low[2] = min (low[2], tin[0])
= min (2, 0) = 0 ; so update low[2]

}



tin [0|1|2|-1]
low [0|1|0|-1]
parent [-1|0|1|-1]
vis [T|T|T|F|F]

Now, when we are returning we do,

step 1: low[node] = min (low[node], low[child]);

low[1] = min (low[1], low[2])
= min (1, 0) = 0

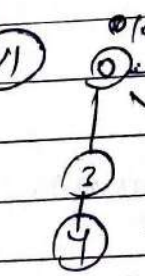
step 2: To check bridge

if (low[neighbour] > tin[node]) then there is bridge.

This means it is only one way to reach neighbour from node

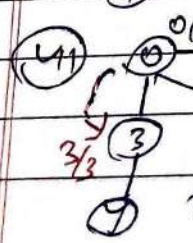
here if (low[2] > tin[1]) ≡ if (0 > 1) (X)

Hence there is no bridge.



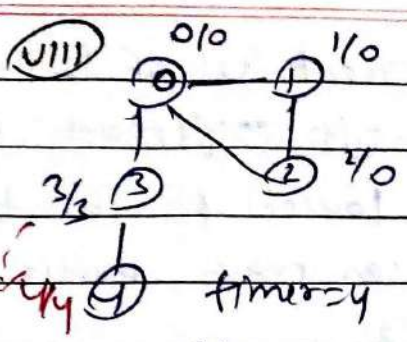
step 1 low[0] = min(low[0], low[1]);
= min (0, 0) = 0.

step 2 low[1] > tin[0] ≡ 0 > 0 (X)



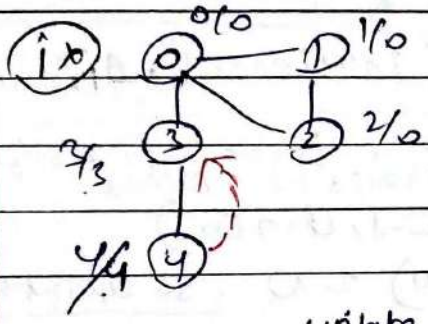
tin [0|1|2|3|-1]
low [0|0|0|3|-1]
parent [-1|0|1|0|-1]
vis [T|T|T|T|F]

(viii)



	0	1	2	3	4
tin	0	1	2	3	4
low	0	0	0	3	4
parent	-1	0	1	0	3
vis	T	T	T	T	T

(ix)



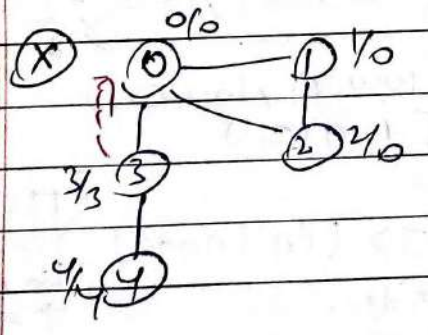
here neighbour of 4 which is 3 is parent of 4 so we will ignore this and return. when will return in 3 then update low & check for bridge.

(1) low[3] = min(low[3], low[4])
= min(3, 4) = 3 (no update)

(2) low[4] > tin[3] ≡ 4 > 3 (T)

hence, there is bridge b/w (3) & (4)

(x)



(1) low[0] = min(low[0], low[3])
= min(0, 3) = 0

(2) low[3] > tin[0] ≡ 3 > 0 (T)

hence, there is bridge b/w (0) and (3)

Code:—

```
#include <bits/stdc++.h>
using namespace std;
void dfs(int node, int parent, vector<int> &tin, vector<int> &low,
         int &timer, vector<int> adj[]){
    vis[node] = 1;
    tin[node] = low[node] = timer++;
```

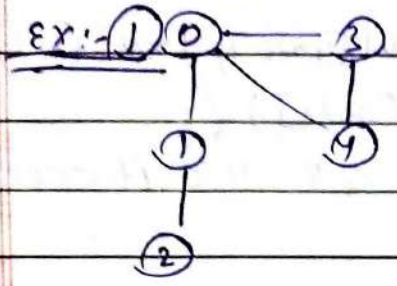
```
for(auto it: adj[node]){
    if(it == parent) continue;
    if(!vis[it]){
        dfs(it, node, vis, tin, low, timer, adj);
        low[node] = min(low[node], low[it]);
        if(low[it] > tin[node]){
            cout<<node<<" "<<it<<endl;
        }
        else{
            low[node] = min(low[node], tin[it]);
        }
    }
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n];
    for(int i=0; i<m; i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> tin(n,-1);
    vector<int> low(n,-1);
    vector<int> vis(n,0);
    int timer=0;
    for(int i=0; i<n; i++){
        if(!vis[i]){
            dfs(i,-1,vis,tin,low,timer,adj);
        }
    }
    return 0;
}
```

Bridge

TC: O(V+E)
SC: O(2V)

(16) Articulation Point :-

- articulation point of a graph is the vertex by removing which graph become disconnected. or, increases the number of connected components.

Ex:- 

Articulation points here are :-
0, 1

Ex:- 2 :-

Articulation points are :- 2,3,4

Approach ① :- Brute Force :- $O(V*(V+E))$

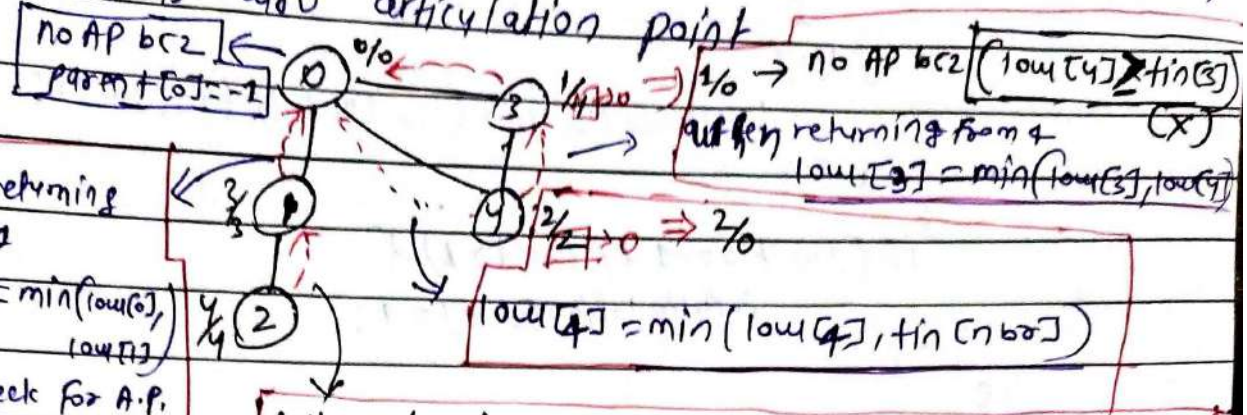
a simple method is to remove all vertices one by one and see if it causes the graph to become disconnected (using BFS/DFS)

Approach ② :- Optimized

all the process will be same as previous question to find bridges in graph. but instead of checking bridges we have to check Articulation point and we use the formula `low[neighbour] >= tin[node] && parent != -1` (Y) instead of checking bridge formula. `low[nbr] > tin[node]` (X)

- this formula says that, there is no any other way by which neighbour can be reached. If lowest of neighbour is at most equal to arriving time of node this means neighbour can be reached only by way of node.
- condition will be false if neighbour lowest time will be less than node arrival time means there is other way before node to reached neighbour
- also parent != -1 it means if node is src node then this will not be articulation point but it is contradict that 0 is source and it is also a A.P.

so to handle the parent case (i.e source node) we will check child of src, if it is greater than one then src is also articulation point



To handle '0' src child of '0' ≥ 2
∴ '0' is A.P

Code:-

```
#include<bits/stdc++.h>
using namespace std;
void dfs(int node, int parent, vector<int> &vis, vector<int> &tin,
         vector<int> &low, int &timer, vector<int> adj[], vector<int> &isArticulation)
{
    vis[node] = 1;
    tin[node] = low[node] = timer++;
    int child = 0;
    for(auto it: adj[node]){
        if(it == parent) continue;
        if(!vis[it]){
            dfs(it, node, vis, tin, low, timer, adj, isArticulation);
            low[node] = min(low[node], low[it]);
            child++;
```

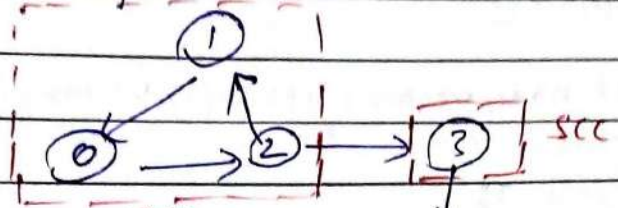
```
        if(low[it] >= tin[node] && parent != -1){
            isArticulation[node] = 1;
        }
    }else{
        low[node] = min(low[node], tin[it]);
    }
}
if(parent == -1 && child > 1){
    isArticulation[node] = 1;
}
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<int> adj[n];
    for(int i=0;i<m;i++){
        int u,v;
        cin>>u>>v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> tin(n,-1);
    vector<int> low(n,-1);
    vector<int> vis(n,0);
    vector<int> isArticulation(n,0);
    int timer=0;
    for(int i=0;i<n;i++){
        if(!vis[i]){
            dfs(i,-1,vis,tin,low,timer,adj,isArticulation);
        }
    }
    for(int i=0;i<n;i++){
        if(isArticulation[i]==1) cout<<i<<endl;
    }
    return 0;
}
```

TC: O(V+E)
SC: O(N)

(17) Kosaraju's Algorithm for strongly connected components (SCC)

SCC is that if we start from any node in a component, we must be able to reach all other nodes in that component. Note that by component here, we mean group of certain nodes in the graph that meet the condition for every node in that component.

Ex:-

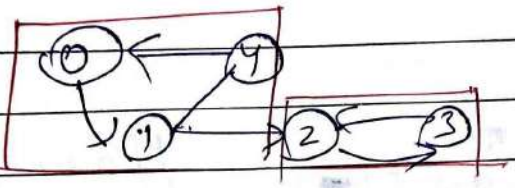


Answer

SCC
0 ↔ 2 ↔ 1
3
4

DFS 0 → 2 → 1
2 → 1 → 0
1 → 0 → 2

Ex



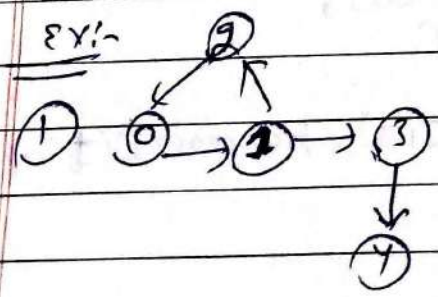
Answer :- SCC
0 ↔ 1 ↔ 4
2 ↔ 3

Intuition :- The idea behind kosaraju's algorithm is to do a DFS in controlled fashion such that we won't be able to go from one scc to another. One DFS call would visit all the nodes in an scc only.

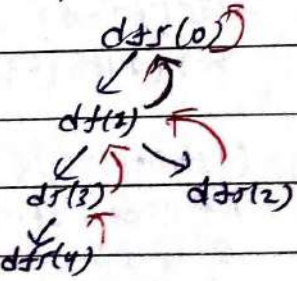
Approach :- (Topological sort)

1. Sort all nodes basis on their finishing time.
2. Transpose graph. reverse all edges of graph.
3. Use tolopogical sort element from step 1 to find SCC using DFS.

Ex:-

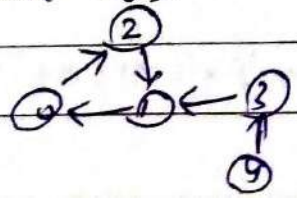


Topological sort



reverse edges.

(ii)



(iii) dfs on topo :- 0 1 2
3
4

Date

Page No. 300.

Code:-

```
#include <bits/stdc++.h>
using namespace std;
void dfs (int node, stack<int> &st, vector <int> &vis, vector<int> adj[]){
    vis[node] = 1;
    for(auto it: adj [node]){
        if (!vis [it]){
            dfs (it, st, vis, adj);
        }
    }
    st.push(node);
}
void revDfs (int node, vector<int> &vis, vector<int> transpose[]){
    cout <<node << " ";
    vis [node ] = 1;
    for (auto it: transpose[node ]){
        if (!vis [it]){
            revDfs (it, vis, transpose);
        }
    }
}
int main(){
    int n=5, m=7;
    vector<int> adj [n+1];
    for(int i=0; i<m; i++){
        int u,v;
        cin >>u>>v;
        adj [u].push_back (v);
    }
    stack <int> st;
    vector <int> vis (n+1,0);
    for (int i=0; i<n+1; i++){
        if(!vis[i]) dfs (i, st, vis, adj);
    }
    vector<int> transpose [n];
    for (int i=1; i<n; i++){
        vis[i]=0;
        for(auto it: adj[i]) transpose [it].push_back(i);
    }
    while (!st.empty()){
        int node = st.top();
        st.pop()
        if (!vis[node]){
            revDfs (node, vis, transpose);
            cout<<endl;
        }
    }
    return 0;
}
```

// 0' based indexing graph

topo sort

transpose

SCC

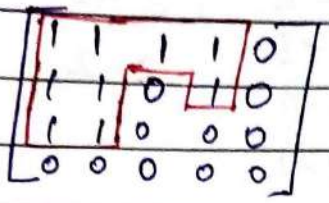
Date

Page No. 351.

(19) No. of island (leetcode 200):

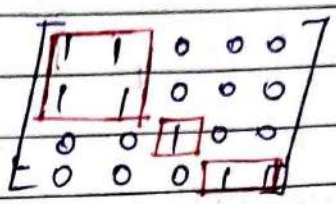
- Given an mxn 2D binary grid which represent a map of '1's (land) and '0's (water). return the no. of island.
- An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.

Ex:-



O/P : 1

Ex:-



O/P : 3

Approach

- For every element if it is '1' make DFS call in 4 direction (x-1, y) ↑, (x, y-1) ←, (x, y+1) →, (x+1, y) ↓ and make '1' to another no. like '2' that indicate this land is already visited.

Code :-

```
class Solution {
    void dfs (vector<vector<char>>& matrix, int x, int y, int r, int c){
        if (x<0 || x>=r || y<0 || y>=c || matrix[x][y] != '1')
            return;    //Boundary case for matrix

        matrix[x][y] = '2';
        dfs (matrix, x+1, y, r, c); //Down
        dfs (matrix, x, y+1, r, c); //right
        dfs (matrix, x-1, y, r, c); //top
        dfs (matrix, x, y-1, r, c); //left
    }
public:
    int numIsland (vector<vector<char>>& grid){
        int row = grid.size();
        if(row==0); return 0;
        int col = grid[0].size();
        int no_of_island = 0;
        for(int i=0; i<row; i++){
            for(int j=0; j<col; j++){
                if (grid[i][j]=='1'){
                    dfs(grid, i, j, row, col);
                    no_of_island++;
                }
            }
        }
        return no_of_island;
    }
};
```

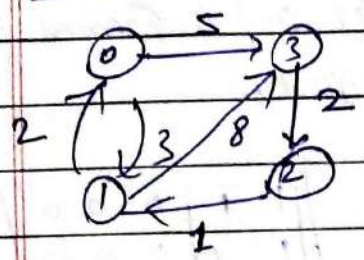
TC: O(m×n)

SC: O(m×n)

(19.) Floyd Warshall algorithm / All pairs shortest path.

- Find all pairs shortest path in a given graph.

Note:- shortest distance of Node to itself will always be zero.



All pairs

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

Ex:- d[1][3] = ~~8~~ 7

Note:- we include each vertex one by one in calculation of shortest path.
as soon as we find a shortest path we update our shortest path value.

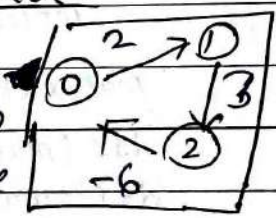
Formula:-

the dist via 'k',

$$d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

* DETECT -VE EDGE WEIGHT CYCLE

- If we have -VE distance from a vertex to itself (ex from 0 to 0) then we have a negative edge weight cycle
- in this case there will never be shortest path from 0 → 0 every time it will go in minus.



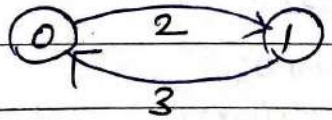
0 → 0 ⇒ 0

0 → 2 → 2 → 0 (-2)

0 → 0 ⇒ -2

* INCLUDING NODE ADJACENT WON'T AFFECT DISTANCE

- if i → k → j if 'k' is adjacent of i i.e it is 'j', ∴ i → j → j



- d[0][1] = 2

then, including vertex-2

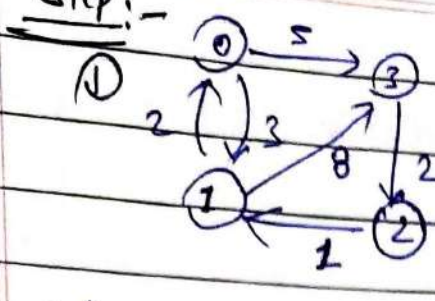
d[0][1] = min(d[0][1], d[0][2] + d[2][1])

= min(d[0][1], d[0][2] + 0) = d[0][1]

Date

Page No. 353,

Step:-



	0	1	2	3
0	0	3	∞	5
1	2	0	∞	8
2	∞	1	0	∞
3	∞	∞	2	0

	0	1	2	3
0	0	3	∞	5
1	2	0	∞	7
2	∞	1	0	∞
3	∞	∞	2	0

Note:- • Digonal all zero (i.e no selfnode)
• To Finite indicate there is no edge.

D^0 i.e for every vertex go via 'k=0' i.e

$d[0][0] = \min(d[0][0], d[0][0]+d[0][0])$

$d[0][1] = \min(d[0][0], d[0][0]+d[0][1])$

$d[0][2] = \min(d[0][0], d[0][0]+d[0][2])$

$d[0][3] = \min(d[0][0], d[0][0]+d[0][3])$

$d[1][0] = \min(d[1][0], d[1][0]+d[0][0]) = \min(2, 2+0) = 2$

$d[1][1] = \min(d[1][1], d[1][0]+d[0][1]) = \min(0, 2+3) = 0$

$d[1][2] = \min(d[1][2], d[1][0]+d[0][2]) = \min(\infty, 2+\infty) = \infty$

$d[1][3] = \min(d[1][3], d[1][0]+d[0][3]) = \min(8, 2+5) = 7$

	0	1	2	3
0	0	3	∞	5
1	2	0	∞	7
2	3	1	0	8
3	∞	∞	2	0

similarly via 'k=1' will update for every vertex.

D^1 i.e for every vertex go via 'k=1'.

Note:- similarly it will go from k=0 to k=3 i.e k=v for every vertex. and update distance.

Thus final update will be.

D^3.

	0	1	2	3
0	0	3	∞	5
1	2	0	∞	7
2	3	1	0	8
3	5	3	2	0

D^4.

	0	1	2	3
0	0	3	7	5
1	2	0	9	7
2	3	1	0	8
3	5	3	2	0

Note:- in final updated matrix check if there is -ve value in diagonal means there is -ve edge cycle bcz the value of node is -ve here.

to reach

Time complexity = $O(V^3)$ / space complexity = $O(V^2)$

Code :-

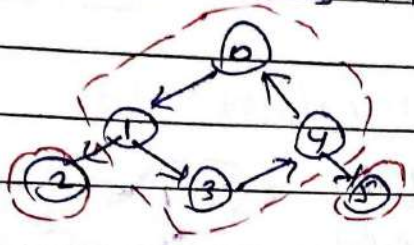
```
void Floyd-Warshall (int graph[V][V])
{
    int dist[V][V];
    for(int i=0; i<V; ++i)
        for(int j=0; j<V; ++j)
            dist[i][j] = graph[i][j];

    for(int k=0; k<V; ++k)
        for(int i=0; i<V; ++i)
            for(int j=0; j<V; ++j)
            {
                if(dist[i][k]==INT_MAX || dist[k][j]==INT_MAX)
                    continue;
                else if(dist[i][k]+dist[k][j]<dist[i][j])
                    dist[i][j] = dist[i][k]+dist[k][j];
            }

    For(int i=0; i<V; ++i)
        if(dist[i][i]<0){
            cout<<" Negative edge weight cycle is present \n");
            return;
        }

    For(int i=1; i<V;++i){
        For(int j=0; j<V;++j)
            cout<<i<<" to "<<j<<" distance is "<<dist[i][j]<<\n;
        cout<<"==========\n";
    }
}
```

Note:- here graph[V][V] is adjacency matrix of graph.

(20) TARJAN's Algorithm (strongly connected component)

- a strongly connected component is that if we start from any node in a component we must be able to reach all other nodes in that component

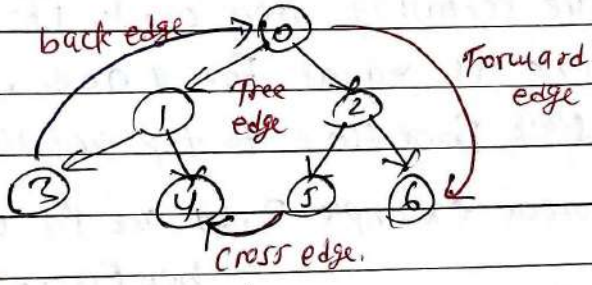
eg:-



there are 3 SCC in this graph

*Note:- Kosaraju's algorithm take O(3(V+E)) time and Tajan's solve this problem in only one loop O(V+E).

*Note:- types of edges in graph.



cross edge:- don't have ancestor-descendent relationship.

Tree edge: (u,v) — parent, child

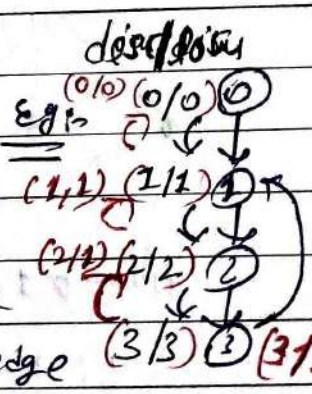
Forward edge: (u,v) — ancestor, descendant

Back edge:-(u,v) — descendant, Ancestor

*low vs disc

- low (node with lowest discovery time)
- disc (discovery time of the node)

eg:-



Note:- when there is a back edge from a node then update that low of node with back edge disc time

- here (3) low will be low[2] = (min(low[2], disc[1]) = 1

Note:- when returning then check low & update with min low time

there (2) low will be update as

low[2] = min(low[2], low[3])
= 1

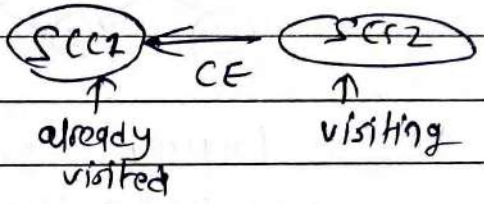
Repeat for all.

* How to identify back edge: & cross edge.

- we take a stack to keep track of node parent in SCC
- if edge is pointing to a visited node already in stack then it is back edge else if is cross edge

* Why cross-edge is not processed.

- we don't want to again cover the already covered SCC
- since it is already visited and not in stack means there is no way to go from SCC1 to SCC2

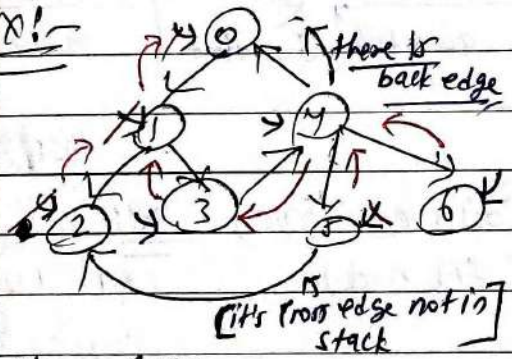


* To find SCC

when we are returning then check if low, time & discovery time is equal for a node. then the node present in stack from these to top are all is one SC

Note:- if previous example 3, 2, 1 are in one SCC.
bcz (low[2] = disc[2])

Ex:-



time = 0 1 2

SCC1 : 2 | SCC2 : 5 | SCC3 : 6

SCC4 : 4 3 1 0

Disc

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1
0	1	2	3	4	5	6

low

0	1	2	3	4	5	6
0	1	2	3	4	5	6
0	1	2	0	0	5	6

Instack

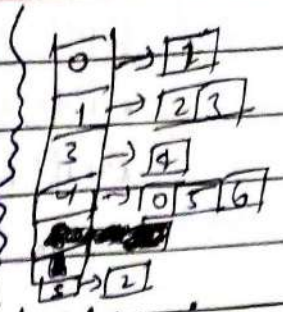
0	1	2	3	4	5	6
F	F	F	F	F	F	F
F	F	F	F	F	F	F

Stack | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Note:- here do not need to ~~check~~ take visited array disc array can be used to check visited.

Time COMPLEXITY = O(V+E) (Single Traversal algo)

```
#include <bits/stdc++.h>
using namespace std;
#define v 7
#define pb push_back
unordered_map<int, vector<int>> adj;
```



(2) `void DFS(int u, vector<int>& disc, vector<int>& low, stack<int>& mystack, vector<bool>& presentInStack)`

```
{
    static int time=0;
    disc[u]=low[u]=time;
    time+=1;
    mystack.push(u);
    presentInStack[u]=true;
    for(int v: adj[u]){
        if(disc[v]==-1){
            DFS(v, disc, low, mystack, presentInStack);
            low[u]=min(low[u], low[v]);
        }
        else if(presentInStack[v])
            low[u]=min(low[u], disc[v]);
    }
    if(low[u]==disc[u]){
        cout<<"scc is: ";
        while(mystack.top()!=u){
            cout<<mystack.top()<<" ";
            presentInStack[mystack.top()]=false;
            mystack.pop();
        }
        cout<<mystack.top()<<"\n";
        presentInStack[mystack.top()]=false;
        mystack.pop();
    }
}
```

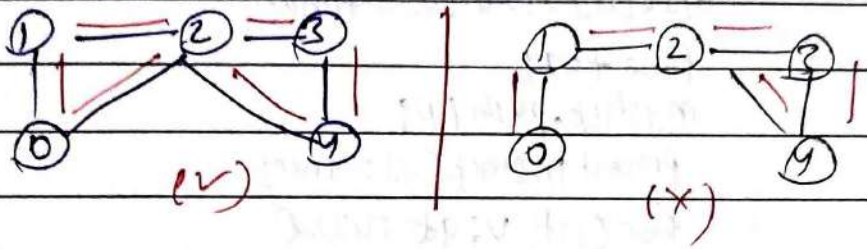
(1) `void findSCCs-Tarjan(){`

```
    vector<int> disc(v,-1), low(v,-1);
    vector<bool> presentInStack(v,false);
    stack<int> mystack;
    for(int i=0; i<v; ++i)
        if(disc[i]==-1)
            DFS(i, disc, low, mystack, presentInStack);
}
```

(21) EULER GRAPH & CIRCUIT

- walk :- Any random traversal in graph.
- trail :- A walk in which no edge is repeated (vertex can be repeated)
- Euler circuit :- A trail which starts & ends at same vertex is called Euler circuit.
- Conditions :-
  i) start = end ii) every edge must be visited only once.

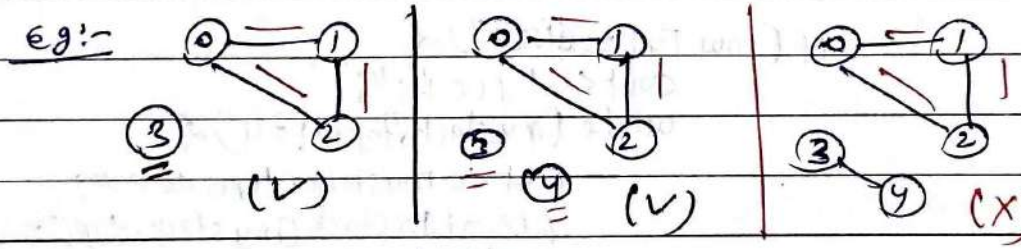
eg:-



(*) EULER GRAPH :- graph having eular circuit.

- All edges in a graph must be present in a single component (with above 2 condition)
- All other component should not have any edge and hence should be of size 1 - vertex only

eg:-



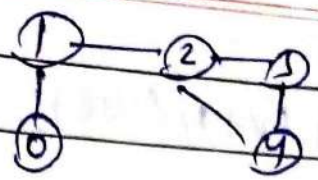
- All vertices should have even degree
- All vertices with non-zero degree are connected in a component. Rest all vertices must have 0-degree
- Graph with no edges ~~is~~ eular graph.

(*) Euler path, SEMI-EULERIAN GRAPH

- Euler path is a path that visits every edge exactly once not necessary (start node = end node)

Semi-Eulerian graph

① every edge visited once ② ~~Start vertex~~ ≠ end vertex



- exactly 2- vertices must have odd edge (start & end v)
- all vertices with non-zero degree are connected.

Algorithm Step:-

(1) Connectivity check → check all edges are present in 1-component only.

① Find a node with degree > 0
If, no node found the eulor graph
else { do DFS and mark all node in component
check if any node with degree > 0 was unvisited
if True → not eulor graph

② Count odd degree nodes (use adjacency list)
count = 0 ⇒ Eulerian graph.
count = 2 ⇒ semi-Eulerian graph.
count > 2 ⇒ not-Eulerian graph.

Time Complexity = O(V+E)

NOTE:- No. of vertices of odd degree in an undirected graph is even.

Code:-

```
#include<bits/stdc++.h>
using namespace std;
#define v 5
#define pb push_back

unordered_map<int, vector<int>> adj;
void DFS(int curr, vector<bool>& visited){
        visited[curr] = true;
        for(auto it: adj[curr])
                if(!visited[it])
                        DFS(it, visited);
}
```

```
bool Connected_Graph(){
    vector<bool> visited (V+1, false);
    int node=-1;
    for(int i=0; i<V; ++i)
        if (adj[i].size()>0){
            node=i;
            break;
        }
    if (node==-1)
        return true;
    DFS (node, visited);
    for(int i=0; i<V; ++i)
        if (visited[i]==false && adj[i].size()>0)
            return false;
    return true;
}

int Find_Euler(){
    if (! Connected_Graph()) return 0;
    int odd=0;
    for(int i=0; i<V; ++i)
        if (adj[i].size()%2==1) odd+=1;
    if(odd>2) return 0;
    return (odd==0)?2:1;
}

void FindEuler_Path_Cycle(){
    int ret=Find_Euler();
    if(ret==0) cout<<"GRAPH is NOT a Euler graph";
    else if(ret==1) cout<<"GRAPH is semi-Eulerian";
    else cout<<"GRAPH is Eulerian";
}
```