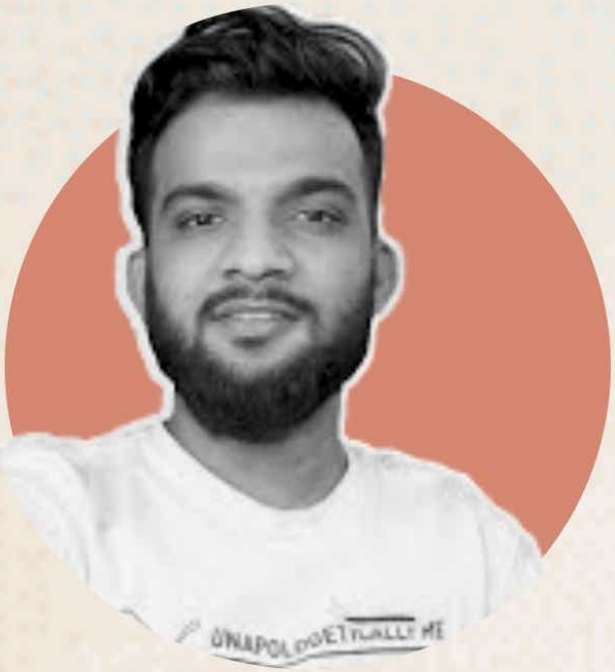
GRAPH DATA STRUCTURE

Handwritten Notes



Raj Vikramaditya **Tapesh Dua**

NOTES

Graph data Structure

Data Structure → have 2 types
- Linear
- Non-Linear

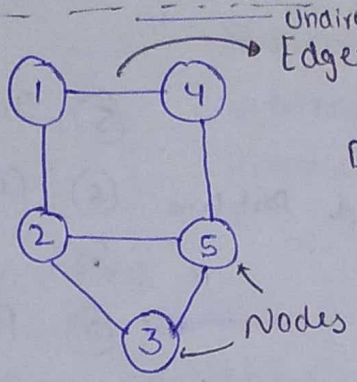
A Non linear ds consisting of nodes that have data and are connect to other nodes through edges.

Tree is a special type of graph with some restrictions.

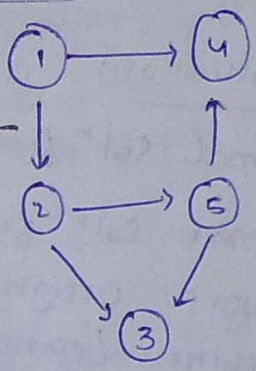
Nodes : AKA vertices, store the data. Numbering of the nodes can be done in any order.

Edge : Two Nodes are connected by a horizontal line. Edge can be directed or undirected.

Representation



Undirected Graph

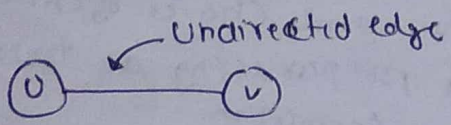


Directed Graph

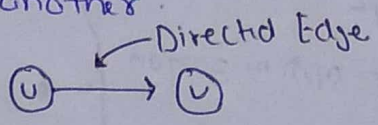
Types of Graph

① Undirected Graph : where edges are bidirectional, x direction.

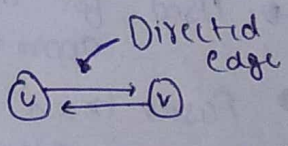
② Directed Graph : All edges are directed from one node to another.



There is an edge b/w U → V and V → U as well.

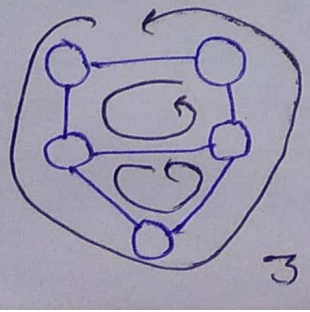


There is an edge b/w U → V

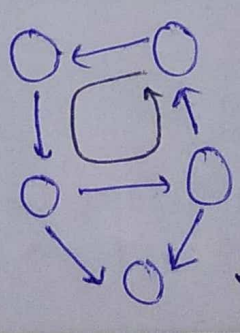


An edge b/w U → V and V → U

Graph doesn't necessarily mean to be an enclosed structure, it can be an open structure as well. A Graph have cycle if it starts from a node and ends at same node.



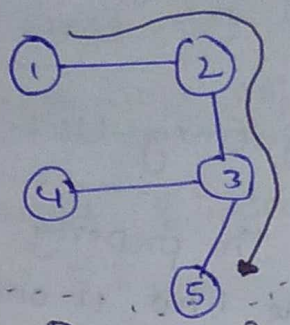
3 cycles → Undirected Cyclic Graph



1 cycle → Directed Cyclic Graph

Directed Acyclic Graph (DAG) : Directed graph with no cycle

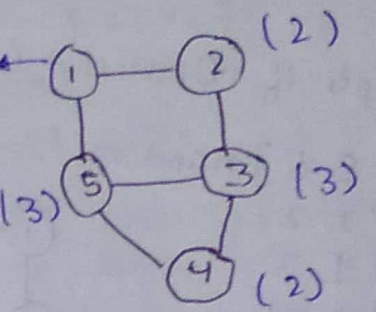
Path in a Graph : Path contains a lot of nodes and each of them is reachable.



1 - 2 - 3 - 5 is a path.
1 - 2 - 3 - 2 - 1 is x path (node x appear twice)
1 - 3 - 5 is x path (no edge b/w (1) and (3))

Degree of a Graph : It is the no. of edges that go inside or outside that node.

- For Undirected Graph
no. of edges attached to a node



* Property: Total degree = 2 x no. of Edges

Because every edge is associated to two nodes.

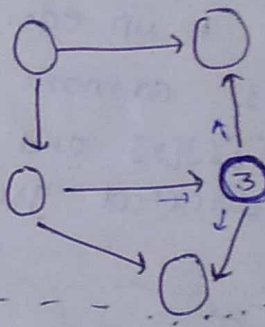
$2 + 2 + 3 + 3 + 2 = 2 \times E$

$12 = 2 \times 6$

$12 = 12$

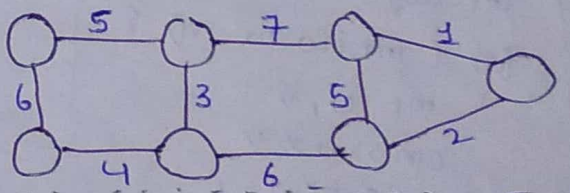
- For directed Graph we have indegree & out degree

↳ Indegree : No. of incoming edges
↳ Out degree : No. of outgoing edges.



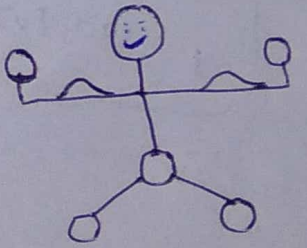
Edge Weight : A Graph may have weights assigned on its edges often referred as Cost of edge

If weights are not assigned then assume the unit weight.



Applications of Graphs

- Analysis of electric circuit
- Shortest routes b/w two places
- Navigation systems like Google maps
- Social media, to store data about each user.



⇒ C++ Representation of Graph

Input format: 1st line : n (number of nodes) m (number of edges)

next m lines : u v → representing an edge b/w u & v.

There is no boundation to number of edges

Representation : ① Adjacency Matrix ② Adjacency Lists.

→ Adj Matrix : It is 2D array of size n×n with the property that adj[u][v] = 1 → If edge b/w u and v

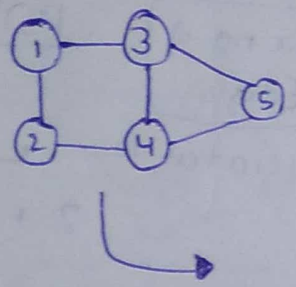
adj[u][v] = 0 → If X edge " u and v

- Example of Undirected Graph

Nodes can be 0 based or 1 based

I/P 5 6] n and m

Edges
1 2
1 3
2 4
3 5
3 4
4 5



For one based → adj matrix size

int adj[n+1][n+1]

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

No edge b/w (5) and (0)

Represent edge b/w (5) and (3)

If there is an edge b/w 1 and 2 ⇒ mark 1 at adj[1][2] and adj[2][1] for undirected graph.

- Code:

```
int n, m
cin>>n>>m
int adj[n+1][m+1]
for ( int i=0 to m) {
    int u, v
    cin>>u>>v
    adj[u][v] = 1
    adj[v][u] = 1   } → x required for directed graph
}
```

Time Complexity : O(m)

Space Complexity : O(n²)

Costly.

Adj Lists [takes less space]

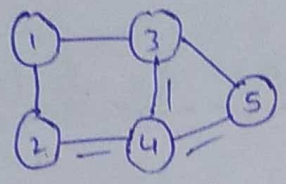
This is node-based representation. We associate node with each a list of nodes adjacent to it.

To create adj list, create array of type vector of size n+1 for 1 based.

`vector<int> adj[n+1];` → Every idx is containing empty vector.

Example of undirected graph.

I/P
5 6
1 2
1 3
2 4
3 4
3 5
4 5



Node 4 has adjacent node 2, 3 and 5

0 → { }
1 → { 3, 2 }
2 → { 1, 4 }
3 → { 1, 4, 5 }
4 → { 2, 3, 5 }
5 → { 3, 4 }

adj stores immediate neighbours in any order.

Code:

```
int n, m
cin >> n >> m
vector<int> adj[n+1]
for (i=0 to m) {
    int u, v
    cin>>u >> v
    adj[u].push_back(v)
    adj[v].push_back(u)   // required for directed graph
}
```

In UG → each edge data appears twice. So space need to represent UG → 2 × E

E = m

	UG	DG
Space Complexity	2 × E	E
Time Complexity	E	E

Weighted Graph representation

For adj Matrix : instead of storing 1 store the weight

UG	DG
int u, v, wt cin >> u >> v >> wt adj[u][v] = wt adj[v][u] = wt	int u, v, wt cin >> u >> v >> wt adj[u][v] = wt

For adj list, store pairs (adjacent Node, edge weight)

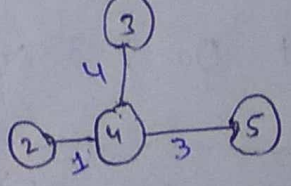
`vector<pair<int, int>> adj[n+1]`

Example:

4 → { (2, 1), (3, 4), (5, 3) }

edge b/w (4) and (2)

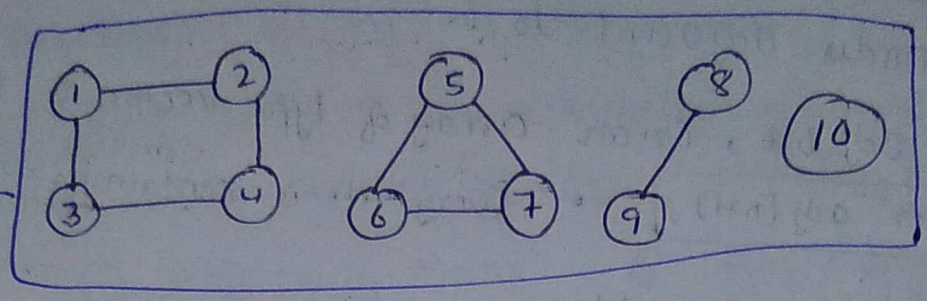
edge weight is 1



Connected Components in graphs

The graph can be broken down into different connected components.

Example



→ Created from a single s/p

Single graph having 10 nodes and 4 different connected components.

Graph Traversal

Any traversal will always use a visited array `//int vis[n+1]//`

```
for (int i=1; i<=10; i++)
    if (!vis[i])
        traversal(i);
```

Why can't we just call traversal(1)

→ Because a graph can have multiple components and traversal algos are designed that they will traverse entire connected portion.

Ex: traversal(1) will traverse only nodes [2, 3, 4] but not connected components.

Breadth First Search (BFS) : Level Order Traversal

Algorithm:

1. Start with a "starting" node and mark it as visited and push into Queue
2. In every iteration, pop out 'node' from Queue and add to result.
3. Go to all unvisited adjacent nodes of 'node', mark them visited and push into queue. (adjacent - access from adj list)
4. Do 2-3 step untill queue becomes empty.

TC $O(N + 2E)$: N nodes, 2E for total degrees as we traverse all adjacent.

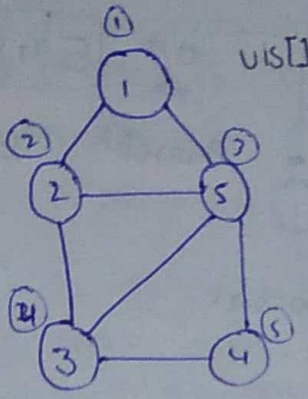
SC $O(N)$: Queue & visited array.

Code:

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

Working of BFS:

0 → { }
1 → { 2, 5 }
2 → { 1, 5, 3 }
3 → { 2, 4, 5 }
4 → { 3, 5 }
5 → { 1, 2, 3, 4 }



vis[]

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

BFS: 1 2 5 3 4

Q: 4 | 3 | 5 | 2 | 1

→ Depth First Search (DFS)

Technique involves the idea of recursion and Backtracking. Goes in Depth.

Approach:

1. Mark the node as visited and push into result.
2. Traverse all adjacent nodes from adjacency list, call dfs for all unvisited adjacent nodes.

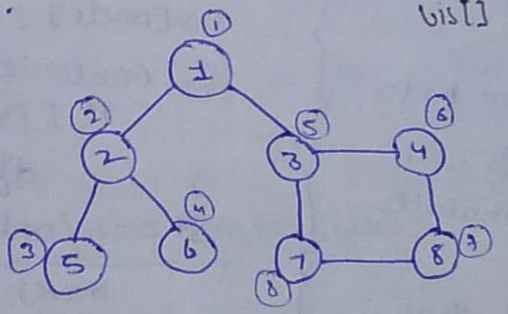
```
vis[node] = 1
dfs.push_back(node)

for(auto it : adj[node]) {
    if (!vis[it]) {
        dfs(it, adj, vis, dfs)
    }
}
```

(Call for adjacent)

TC : O(N + 2E)
SC : O(N) Stack space + vis array

Working of Dfs:

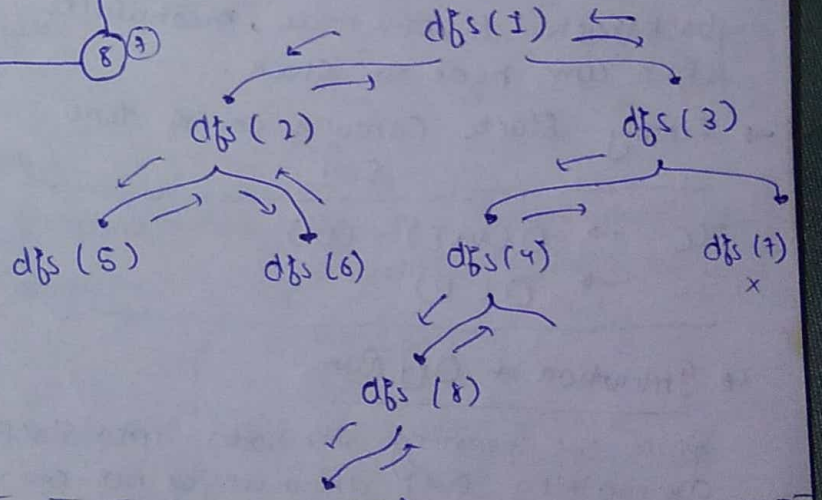
0 → { 3 }
1 → { 2, 3 }
2 → { 1, 5, 6 }
3 → { 1, 4, 7 }
4 → { 3, 8 }
5 → { 2 }
6 → { 2 }
7 → { 3, 8 }
8 → { 4, 7 }



vis[]

0	1	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1	1

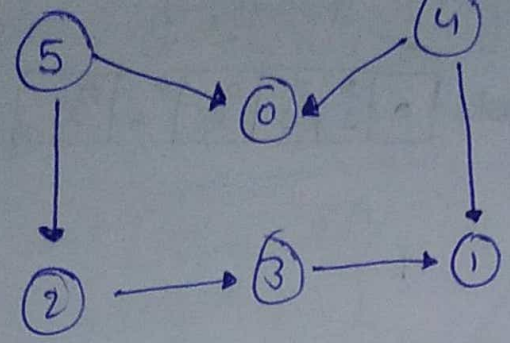
DFS : 1 2 5 6 3 4 8 7



→ Topological Sort Algorithm

In Topological Sorting, node u will always appear before node v if there is a directed edge from node u towards node v [u → v]

Example



I/P

O/P 5 4 2 3 1 0 ← This is one of them.

A Graph may have multiple topo sortings.

Another Possible Result

4 5 2 3 1 0

Edge 5 → 0 : 5 appear befor 0 in Top sort

4 → 0 : 4 " " 0 " "

4 → 1 : 4 " " 1 " "

5 → 2 : 5 " " 2 " "

2 → 3 : 2 " " 3 " "

3 → 1 : 3 " " 1 " "

Toposort only exist in DAG, why?

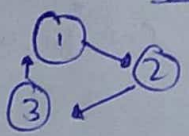
① if edges are Undirected: then there is edge from u → v and v → u as well. Then ordering is practicon impossible that u apears befor v & v appears bfore u

② if graph has Cycle : for 1 → 2 : 1 apears befor 2

2 → 3 : 2 " " 3

3 → 1 : 3 " " 1

} x possible



1 2 3 but 3 should be befor 1

Approach Using DFS

① mark curr node as visited

② Call dfs for all its adjacent nodes

③ After visiting all adj nodes, dfs will backtrack to prv node, meanwhile push curr node to stack.

→ Finally Stack Contains one of topo sort.

TC → O(V+E) + O(V)

SC → O(V)

Code: dfs() O(V+E)

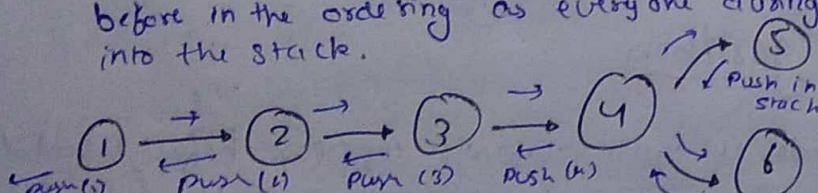
```
vis[node] = 1
for (auto it : adj[nod]) {
    if (!vis[it])
        dfs(it, vis, st, adj)
}
st.push(node);
```

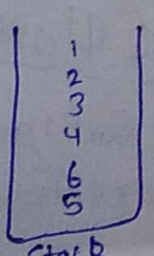
main() O(V)

```
while (!st.empty())
    ans.push_back(st.top())
    st.pop()
```

↳ Intuition + Dry Run

Since we inserting the nodes into stack after completion of traversals we are making sure, there will be no one who appears afterward but may come before in the ordering as everyone during traversal would have been inserted into the stack.





Stack

Topo sort →

1 2 3 4 6 5

Approach using BFS

Using Kahn's Algorithm

- We need arry to store indegree of all nodes. (indegree = no of incoming edgs.)

① Calc indegree, itrate through adj list for every node u → v
indegree[v]++

② There will always be a node with idegree 0. push all node with 0 indegree into queue.

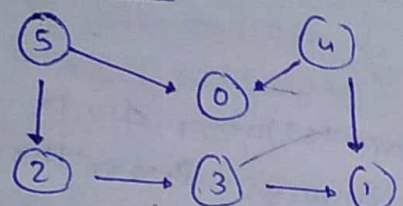
③ Itratively, pop a node from queue add it to result, and decrement indegree of its adjacent nodes.

④ if any node's indegree becomes 0 push into queue.

TC : O(V+E) (Same as BFS)
SC : O(V)

```
int indegree[V] = {0}
for (i=0 to V){
    for (auto it : adj[i]){ ①
        indegree[it]++
}}
queue<int> q
for (i=0 to V){
    if (indegree[i]==0){ ②
        q.push(i)
}}
vector<int> ans;
while (!q.empty()){
    node = q.front()
    q.pop()                ③
    ans.push_back(node)
    for (auto it : adj[node]){
        indegree[it]--
④       if(indegree[i]==0){
            q.push(it)
}}}
```

Dry run:



indegree[]

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

New indegree

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

-1

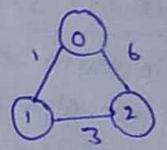
Queue: 5, 4 → 5, 4

ans = { 4, ... }

Dijkstra's Algorithm [To find shortest distance of all vertex from source vertex S]

Can't use when Graph Contains negative weight cycle.

Example



S = 2 O/P

0	1	2
4	3	0

- Initial Config:

PQ : Stores pair {dist, node} [min-heap]
↑ curr dist from source to node.

Dist Array: Initialize with all value ∞ initially and Source dist as 0.

- Algo:

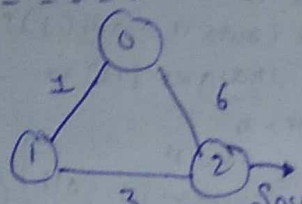
1. Push Source node into PQ with its distance as 0.
2. For every node at top of PQ, look for its adjacent nodes.
3. If curr reachable dist of adj node is better than prv distance stored in distance arry. update the distance and push into PQ.

Code:

```
PQ < pair, greater<pair>> PQ
vector<int> dist(V, Int-max)
dist[S] = 0
PQ.push({0, S})                ①
while (!PQ.empty()){
②   node = pq.top().second
    dis = pq.top().first
    pq.pop()
    for (auto it : adj[node]){
        v = it[0], w = it[1]
③       if (dis+w < dist[v]){
            dist[v] = dis+w
            PQ.push({dis+w, v})
}}}
return dist;
```

A node with a lower distance would be at the top of PQ as opposed to a node with higher distance. By using Step2, we would get the minimum distance from source.

Dry run:



dis[] | ~~4~~ 4 | ~~6~~ 3 | 0 |
0 1 2

PQ {dist, node}
{0, 2}

Step 1

From (2) we can reach at (cur dis + cst)
(0) with dis 0+6 = 6
(1) with dis 0+3 = 3
both are less than ∞ so push into PQ

Step 2 pop {3, 1}

{3, 1}
{6, 0}
~~{0, 2}~~

From (1) we can reach
(2) 3 + 3 = 6 (we can reach 2 at cost 0) 6 is not < 0
(0) 3 + 1 = 4 (4 is less than 6 (prv cost to reach 0))
↑ update this and push into PQ

No point in keeping {6,0} when we get {4,0} which is less. Keeping both in implementation

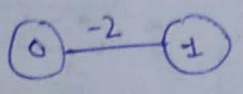
{4, 0}
~~{3, 1}~~
{6, 0}
~~{0, 2}~~

Step 3 → No change will happen

Output → | 4 | 3 | 0 |
0 1 2

Dijkstra's Algo → x valid for neg cycles. Why?

Example



dis[] | ~~0~~ -4 | ~~-2~~ |
0 1

Both the nodes are updated each time we come to them, due to neg weight of '-2' which total distance to node always lesser than prv value, ∴ due to inaccurate results, graph having neg weigh cycles will work.

→ Using Set in C++ STL

We need minimum dist node from top of PQ, we also get it using set as set stores in sorted order.

* Only diff b/w PQ and Set is that in a set we can get if there exists a pair with same node by greater distance than curr, as there will be no point in keeping that node into set. So we simply delete the element.

```
set<pair<int,int>> st
vector<int> dist(v, INT_MAX)
st.insert({0,s})
dist[s] = 0
while (!st.empty()) {          // Run for O(V)
    node = (*st.begin()).second
    dis = "   ".first
    st.erase(it)               // popping some
    for (auto it : adj[node]) {   // Traversing adjacent
        v = it[0], w = it[1]
        if (dis + w < dist[v]) {
            if (dis[v] != INTMAX)
                st.erase({dist[v], v})
            dist[v] = dis + w
            st.insert({dist[v], v})
        }
    }
}
return dist
```

Posting some

Difference

Make it little efficient than PQ.

TC → O(V × (pop + (adjacents × push)))
O(V × (log heapsize + (E × log heapsize)))
O(V × ((V) + ((V-1) + 1)))
O(V² × log(heapsize))
O(V² × 2 log V)
O(E × 2 log V) ≅ O(E log V)

E = V-1
E = V-1

TC → O(E log V) | E = no. of edges
SC → O(E + V) | V = no. of vertices

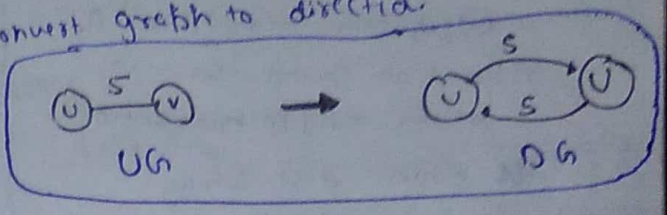
Bellman Ford Algorithm: [Finds the shortest path of all nodes from a source node]

↓ Can handle negative cycles.

Dijkstra → x handle neg edges & cycles

→ Only applicable for directed graphs

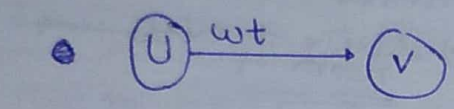
To apply this algo to undirected graph convert graph to directed.



Intution:

Relax all the edges for N-1 times sequentially. After N-1 times, we should have minimized the distance to every node

- What is the relaxation of edges?



Consider this graph with dist[u], dist[v] and wt

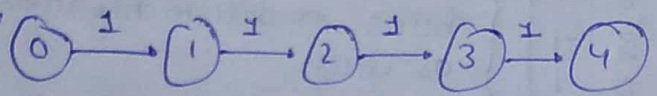
Dist[u]: Shortest distance to reach node u

Dist[v]: " " " " node v

↳ can be ∞

If dist to reach v through u i.e (dist[u] + wt) is smaller than dist[v] update the value of dist[v] with dist[u] + wt → This process of updating distance is called the relaxation of edges.

- Why do we need exact N-1 iterations?



dis[0] = 0

Given order of edges

u	v	wt
3	4	1
2	3	1
1	2	1
0	1	1

Checking in each iteration

dist[3] + 1 < dist[4]

dist[2] + 1 < dist[3]

dist[1] + 1 < dist[2]

dist[0] + 1 < dist[1]

At 1st iteration dist[1] will be updated

dis[2] → in 2nd iter. and so on.

we need total 4 itr. (i.e N-1) to minimize all.

→ Graph of N nodes, need atmost N-1 edges to reach from 1st to last node.

∴ It is impossible to draw a graph that takes more than N-1 edges to reach any node.

- How to detect a negative cycle in the graph?

Acc to our intuition, we should have minimized all the distances within N-1 iterations (means after N-1 → No relaxation is possible)

To check if existing of neg cycle, we will relax edges one more time after the completion of N-1 iterations. If found any relaxation on Nth iteration, we can conclude that graph has neg cycle.

Algo:

1. distance of source node to 0 and rest infinity.
2. Run a loop for N-1 times.
3. Inside, try to relax every given edge.
4. After loop, try to relax all edges one more time. If relaxation found possible return -1 (minimization not possible)
5. Else return dist array

TC : O(V * E)

SC : O(V)

V → num of nodes

E → num of Edges

Code:

```
vector<int> dist(V, Intmax)                    (1)
dist[S] = 0
for (i = 0 to V-1) {                           (2)
    for (auto it : edges) {                    (3)
        u = it[0], v = it[1], wt = it[2]
        if (dist[u] != Intmax and
            dist[u] + wt < dist[v]) {
            dist[v] = dist[u] + wt
    }
}
for (auto it : edges) {                        (4)
    u = it[0]  v = it[1]  wt = it[2]
    if (dist[u] != Intmax and
        dist[u] + wt < dist[v]) {
        return -1
    }
}
return dist
```

→ Floyd Warshall Algorithm [To find shortest path b/w every pair of vertices in weighted directed graph]

- Matrix[I][J] = weight if edge (I) → (J)
- Matrix[I][J] = -1 if no edge.

I/P
$$\begin{bmatrix} 0 & 2 & -1 & -1 \\ 1 & 0 & 3 & -1 \\ -1 & -1 & 0 & -1 \\ 3 & 5 & 4 & 0 \end{bmatrix}$$

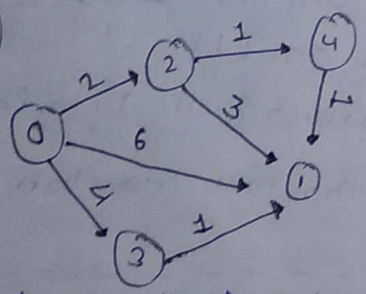
O/P
$$\begin{bmatrix} 0 & 2 & 5 & -1 \\ 1 & 0 & 3 & -1 \\ -1 & -1 & 0 & -1 \\ 3 & 5 & 4 & 0 \end{bmatrix}$$

Final matrix storing shortest Distances. Ex, mat[i][j] storing shortest distance from i to j

Floyd Warshall : multi-source shortest path algo and it helps to detect negative cycle as well.

→ need to check every possible path going via each possible node.

Example



Here we are finding the shortest path from i to j; going via every possible k

We need to find shortest path b/w 0 and 1
Consider path via all nodes.

(0 → 1) = 6

via 2 (0 → 2) + (2 → 1) = 5

via 3 (0 → 3) + (3 → 1) = 5

via 4 (0 → 4) + (4 → 1) = 4 ← min ∴ dist[0][1] = 4

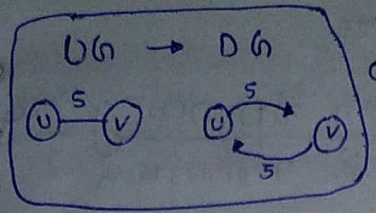
(K)

(0 → 2) + (2 → 4)

Formula

matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])

i → source
J → destination
K → node via we are reaching from i to j.



* Can be used in UG after converting it into DG.

Intuition: To check all possible paths b/w every pair of nodes and to choose the shortest one.

Approach:

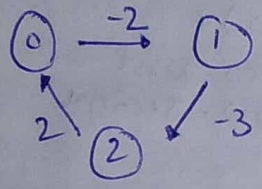
1. Run a loop from 0 to V. In the Kth iteration → check the path via node K for every possible pair of nodes.
2. Inside this loop, there will be two nested loops for generating every possible pairs.
3. Apply the formula inside the nested loop

TC : $O(V^3)$ → 3 nested loops
SC : $O(V^2)$ → storing adj matrix

-1 to inmax

```
for (i=0 to n)
    for (j=0 to n)
        if (mat[i][j] == -1)
            mat[i][j] = Intmax
    if (i==j) mat[i][j] = 0   // distance of node to itself is 0

for (k=0 to n)
    for (i=0 to n)
        for (j=0 to n)
            mat[i][j] = min(mat[i][j], mat[i][k]+mat[k][j])   // formula
```

How to detect negative cycle?



We know that cost of reaching node why itself must be 0.

← Here, if we try to reach node 0 from itself we can follow the path: 0 → 1 → 2 → 0 then cost will be -3 < 0

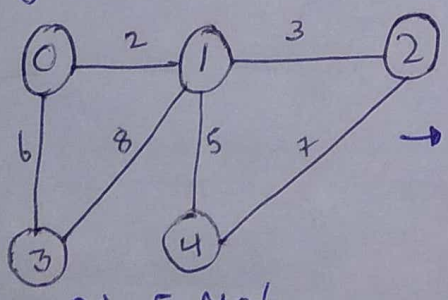
So, if we find that the cost of reaching any node from itself is less than 0, we can conclude the graph has neg cycle.

What will happen if we you use dijkstra's instead of Floyd warshall?

1. → x handle neg cycles → will give TLE
2. → if no negative cycles then, TC of using dijkstra for each source will be $O(V*(E \log V))$ which is less than (V^3) used by floyd warshall.

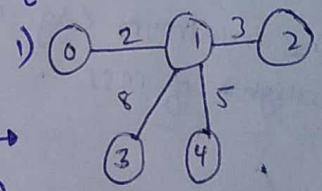
→ Minimum Spanning Tree

A spanning tree is a tree in which we have N nodes and N-1 edges and all nodes are reachable from each other.



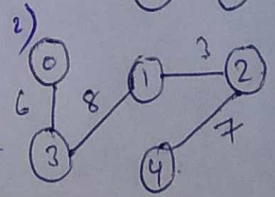
N = 5 M = 6

→ Spanning Trees →

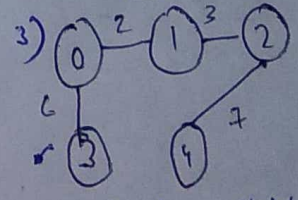


n = 5, m = 4
Sum of edge weight = 18

n = 5, m = 4 ← Sum of edge wt → 24



n = 5, m = 4

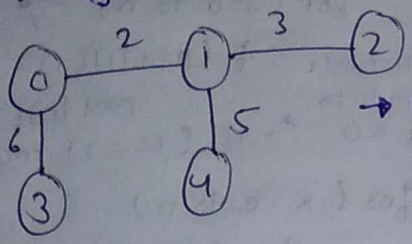


Sum of edge weight = 18

* A graph can have more than 1 spanning trees.

Among all possible trees of a graph, the minimum spanning tree is the one for which the sum of edge weights is minimum

MST of pw graph:-



→ Sum of edge weight = 16 ← Minimum

→ Prim's Algorithm [Used to find MST and sum of edges in MST]

Approach:

We need PQ (min-heap) storing {edge w and node}. To get mst and not only the sum store {edge w, adj node, parent node}
→ vis array, sum = 0, MST array to store.

① push {0, 0, -1} in PQ for (source = 0)

② One by one pop one elemnt from PQ, Check if node is vis continue to the nxt element, if not visited then mark as visited, add weight to sum and store {parent and curr node} in MST.

③ Thn for every adj node, if unvisited store in PQ.

Code:

```
PQ < pair<i,i>, greater<pair<i,i>>> pq
vector<int> vis(V, 0)
                                vect<pair<i,i>>
pq push({0,0})                  mst();
int sum = 0
while (!pq empty()) {
    wt node = pq top().first
    node wt = pq top().second
    pq.pop()
    if (vis[node]) continue
    vis[node] = 1
    sum += wt  → push into MST array if using.
    for (auto it : adj[node]){
        int adjNode = it[0]
        int edw = it[1]
        if (!vis[adjNode]) {
            pq push({edw, adjNo})
    } } }
return sum
```

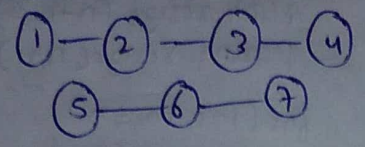
TC : O(E log E)
At most E iterations in (PQ) to become empty

SC : O(E+V)

Intuition: Intuition of this algo is the greedy technique used for every node If we Carefully Observe, for every node, we are greedy selecting its unvisited adj node with the minimum edge weight. Doing so for every node, we can get sum of all edge weights of MST.

⇒ Disjoint Set Data Structure → very imp in entire graphs

Need of Disjoint Set DS?



→ To check if (1) and (5) are in same component or not

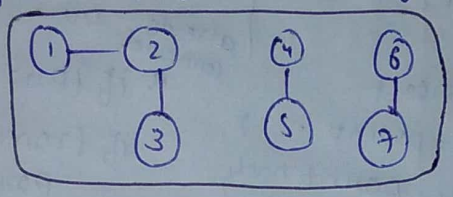
→ we can use DFS / BFS to traverse and check which will cost TC O(N+E)

But using Disjoint set we can solve this in constant time.

Dynamic Graph: A Graph that keeps on changing its configuration.

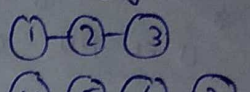
(1,2)
(2,3)
(4,5)
(6,7)
(5,6)
(3,7)

Now if i start adding these edges, in each step graph structure will change

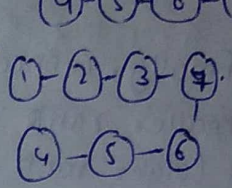


upto (6,7)

After (5,6) →



After (6,7) →



Dsj Set → Generally used for Dynamic Graph

Dsj Set → if we try to figure out whether u and v belong to some component or not. (get in constant time)

Functionalities of Disjoint Set DS:

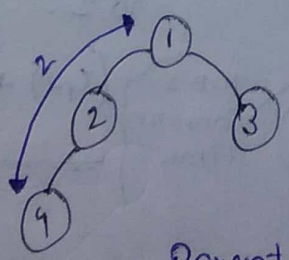
↳ (2)

1) → Finding the parent for a particular node (findPar())
2) → Union [basically adds an edge b/w two nodes]
   - ↳ by rank
   - ↳ by size

Union By Rank

Rank → Rank of a node generally refers to the distance (no. of nodes incl the leaf node) b/w the farthest leaf node and current node.
Rank includes all the nodes below curr node.

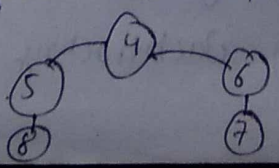
Example:



Rank (1) is 2

Ultimate parent → Parent is right above node, but ultimate parent is topmost node or the root node.

Example:



Parent of (8) is 5
Ultimate Parent of (8) is 4

Implementation using Union By Rank

Need 2 array of size N (no. of nodes)

① Rank[] = {0}

② Parent[] = parent[i]=i → initially every node is parent of itself.

Steps:

→ Union requires two nodes (u and v), then we will find ultimate parent of u and v.

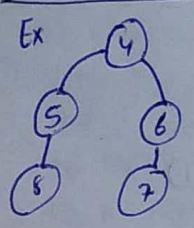
Let's say u → pu (Ultimate parent) and v → pv (Ultimate parent), Find the rank of pu and pv.

→ Finally, connect ultimate parent with a smaller rank to other ultimate parent with a larger rank.

If ranks are equal, we can ~~Correct~~ Connect any parent to other parent.

→ Increase the rank by one for the parent node to whom we have connected other one.

Why Ultimate Parent?



Ex

If we check from right above point then ⑤ and ⑦ belongs to diff component as their parents are diff.

That's why we find Ultimate parent which same for ⑤ and ⑦.

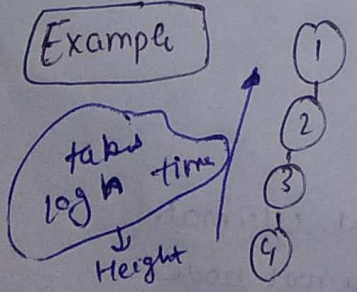
If we find Uparent using recursion of each query, it will end up taking O(log N) TC. But we need constant time. So we'll use

Path Compression.

↓

Connecting each node in a particular path to its Ultimate parent refers to path Compression.

Example



* we are not changing rank during path compression

Approach to findUPar() : using Backtracking

```
class DisjointSet {
    vector<int> rank, parent
Public:
    DisjointSet(int n) {
        rank.resize(n+1, 0)
        parent.resize(n+1)
        for(i=0 to ≤n)
            parent[i] = i
    }
};
```

rank[]: 1 2 3 4 5 6 7 → 0 0 0 0 0 0 0

parent[]: 1 2 3 4 5 6 7 → 1 2 3 4 5 6 7

```
void UnionbyRank(int u, int v)
    pu = findPar(u)
    pv = findPar(v)
    if(pu == pv) return          // already connected
    if(rank[pu] < rank[pv])
        parent[pu] = pv
    else if(rank[pv] < rank[pu])
        parent[pv] = pu
    else {                       // when both have equal rank, connect with anyone.
        parent[pv] = pu
        rank[pu]++
    }
}
```

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

Actual TC of UnionbyRank and findUPar() is $O(4\alpha)$ which is very small & close to 1.

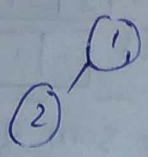
(4α) → long mathematical derivation.

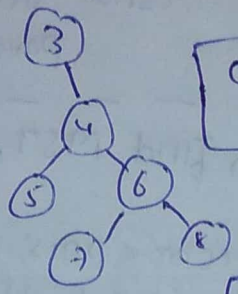
① Base case: if node == parent[node]
return the node.

② Call the findparent of parent of node and while backtracking will update the parent of curr node with returned value.

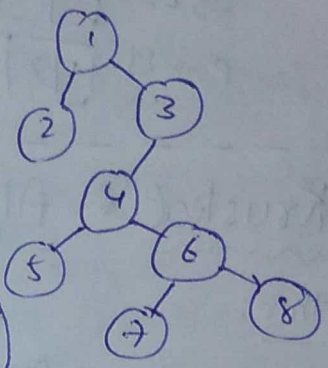
Why we are connecting smaller rank to larger rank?

Example

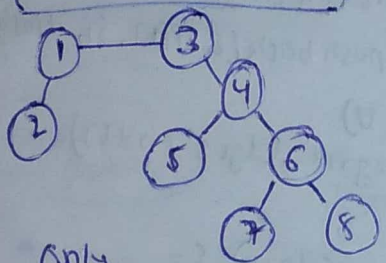




Connecting larger to smaller



Connecting smaller with larger.



Only find UParent of ① and ②

Path compression time ↑

to find UParent of 3, 4, 5, 6, 7, 8

After applying path compression the rank of the graphs becomes distorted.

Rather than storing, we can store the size.

→ Union by Size

Here we need a size array not rank[].

↓

Storing the size of a particular node

← size[i] = size of component from node i.

Initialized with 1.

→ We will connect the ultimate parent with a smaller size to the other ultimate parent with a larger size.

But if the size of the two is equal we can connect any parent to the other parent.

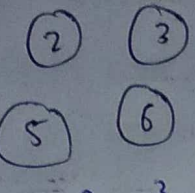
→ Increment the size of whom other compononent is connected.

```
void UnionbySize (u, v) {
    pu = findUPar(u)
    pv =      "     (v)
    if (pu == pv) return
    if (size[pu] < size[pv]) {
        parent[pu] = pv
        size[pv] += size[pu]
    }
    else {
        parent[pv] = u
        size[pu] += size[pv]
    }
}
```

Example

Initial Config.

{1,2}
{2,3}
{4,5}
{6,7}
{5,6}



Sz[]

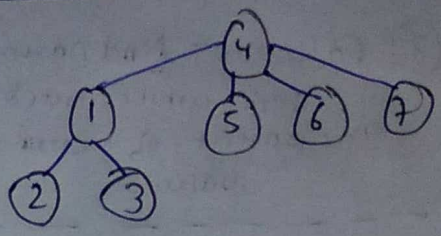
1	2	3	4	5	6	7
1	1	1	1	1	1	1

parr[]

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

Final Config.

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



size[]

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

parr[]

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

Kruskal's Algorithm [To find MST or sum of weights of MST]

Algorithm:

1. Store the edges like this {wt, {u, v}} then sort in ascending order according to the weight. So that we pick min edge first
2. Iterate over all edges
   - for node u and v, check if their Ultimate Parents are same or not.
   - If same → do nothing
   - If different → union(u and v) and add our weight to MST sum.

```
vector<ppi> edges
for (i=0 to V)
    for (auto it: adj[i])
        edges.push_back({it[1], {i, it[0]}})

DisjointSet ds(V)
sort(edges.begin(), edges.end())
int sum = 0
for (auto it : edges) {
    wt = it.first
    u = it.second.first
    v = it.second.second
    if (ds.findUPar(u) ≠ ds.findUPar(v)) {
        sum += wt
        ds.unionBySize(u,v)
    }
}
return sum
```

TC: O(N+E) + O(E log E) + O(E * 4α * 2)
- O(N+E): extracting edge
- O(E log E): sorting
- O(E * 4α * 2): disjoint ops in loop to E times.

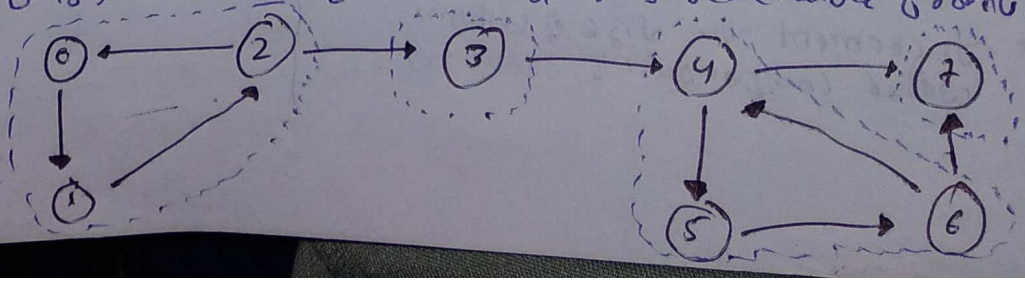
SC: O(N+N+E)
- E → to store edges
- N+N → parent + size[]

Strongly Connected Component - Kosaraju's Algorithm

A component is called a strongly connected component (SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u.

4 SCCs

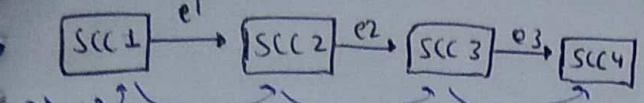
0	1	2	3
4	5	6	7



Kosaraju's Algorithm

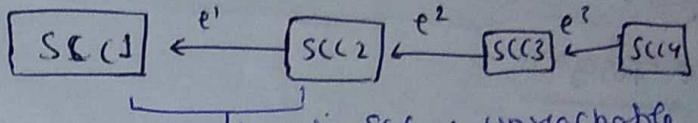
Thought Process If we start simple DFS from source 0, we end up visiting all nodes. So → impossible to differentiate b/w SCCs.

Org graph

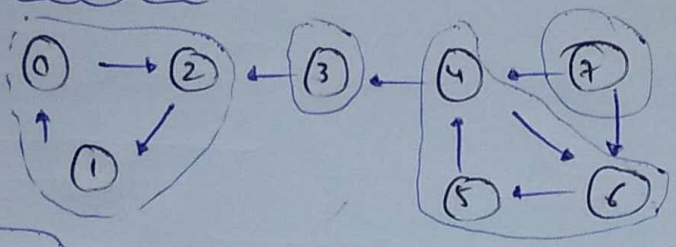


→ If we reverse the edges. Node 0 will only visit the nodes of its SCC 1.

→ Similary node 3 will visit only nodes of SCC 2



→ Now graph



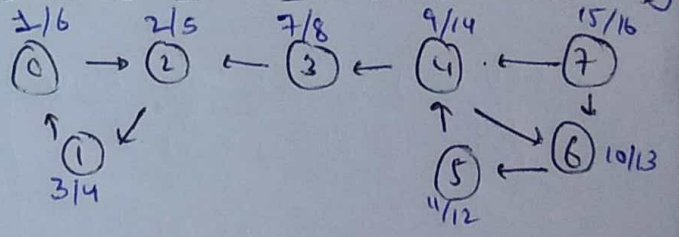
→ After reversing all the edges nodes of same SCC are reachable but nodes of diff SCC are not reachable if we start DFS from source 0.

Another problem Now we can't move from SCC 1 to SCC 2 .... but we can move from SCC 4 to SCC 3 to SCC 2 to SCC 1.

If our source node (0) lies in SCC 4, we again end up visiting all SCCs.

To solve this we have to sort all the vertices in decreasing order on the basis of their finishing time of DFS

Concept of start and finish time



Order : 7 4 6 5 3 0 2 1

Now make a call to dfs in this order.

[Same Algo as Toposort DFS]

Approach:

1. Sort all the nodes according to their finish time.
2. Reverse all the edges of entire graph
3. Perform DFS and count the no. of diff DFS calls to get the no. of SCC.

TC : O(V+E) 2 DFS + Reverse

SC : O(V+E) 2×V for Stack and visited array
V+E → Rev adj

Code:

```
vector<int> vis(V,0), stack<int>st
for (i=0 to V){                    (1)
    if (!vis[i]) { dfs1(i, vis, adj, st)}
}
vector<int> rev-adj
for (i=0 to V){                    (2)
    vis[i]=0
    for (auto it: adj[i]){
        rev adj[it].push back (i)}
}
int SCC=0
while (!st.empty()) {              (3)
    node = st.top() st.pop()
    if (!vis[node]) { SCC++, dfs2(node)
}
return SCC
```

```
dfs1 ( ) {
    vis[node] = 1
    for (aut it : adj [node]) {
        if (!vis [it]) {
            dfs(it)
        }
    }
St.push ( node);
```

→ x in DFS2*