

GRAPH

1) BFS - Breadth First Search $\rightarrow O(E)$

```
void bfs(vector<vector<int>>& adj, int v) {
```

```
    vector<bool> vis(v, 0);
```

```
    queue<int> q;
```

```
    q.push(src); vis[src] = 1;
```

```
    while(!q.empty()) {
```

```
        int u = q.front(); q.pop();
```

```
        cout << u << " ";
```

```
        for(int nbr : adj[u])
```

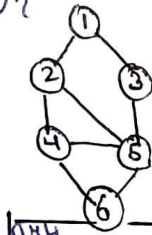
```
        {
```

```
            if(vis[nbr]) continue;
```

```
            q.push(nbr);
```

```
            vis[nbr] = 1;
```

```
        }
```



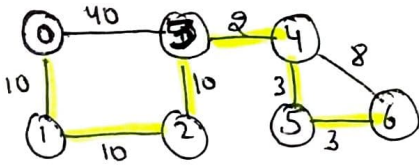
ans

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

traversal

1 2 3 4 5 6

2) PRIMS \rightarrow Algo for MST \rightarrow Greedy Algo \rightarrow remains connected while calc. ans. $\rightarrow O(E \log V)$



ans = 0 + 10 + 10 + 10 + 2 + 3 + 3

Order of Removal

1	(0, 0)
2	(1, 10)
3	(2, 10)
4	(3, 10)
5	(4, 2)
6	(5, 3)
7	(6, 3)

vis

```
int prims(vector<vector<pp>>& adj, int v, int src) {
```

```
    priority_queue<pp, vector<pp>, cmp> pq;
```

```
    pq.push({src, 0}); // node, cost
```

```
    while(!pq.empty()) {
```

```
        pp p = pq.top(); pq.pop();
```

```
        if(vis[p.first]) continue;
```

```
        vis[p.first] = 1
```

```
        ans += p.second;
```

```
        for(auto nbr : adj[p.first])
```

```
        {
```

```
            if(vis[nbr.first] == 0) pq.push({nbr.first, nbr.second});
```

```
        }
```

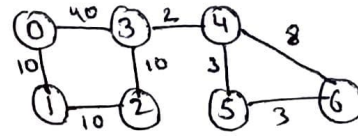
```
    }
```

```
    return ans;
```

3) Dijkstra

only +ve edge weight

\rightarrow Shortest path from source to all vertices
 \rightarrow Greedy
 $\rightarrow O(E \log V)$



```
void dijkstra(vector<vector<pp>>& adj, int v, int src) {
```

```
    priority_queue<pp, vector<pp>, cmp> pq;
```

```
    pq.push({src, 0});
```

```
    vector<bool> vis(v, 0);
```

```
    vector<int> dis(n, INT_MAX); // shortest distance
```

```
    vector<int> par(n); // 1st vertex ka parent
```

```
    dis[src] = 0; par[src] = -1;
```

```
    while(!pq.empty())
```

```
    {
```

```
        pp p = pq.top(); pq.pop();
```

```
        int nod = p.first, cost = p.second;
```

```
        if(vis[nod]) continue;
```

```
        vis[nod] = 1;
```

```
        for(auto nbr : adj[nod])
```

```
        {
```

```
            int nbrnode = nbr.first, nbrcost = nbr.second;
```

```
            if(vis[nbrnode]) continue;
```

```
            pq.push({nbrnode, cost + nbrcost});
```

```
            if(dis[nbrnode] > dis[nod] + nbrcost) {
```

```
                dis[nbrnode] = dis[nod] + nbrcost;
```

```
                par[nbrnode] = nod;
```

```
            }
```

```
        }
```

\rightarrow dis vector contains distance from source to i-th node

\rightarrow par contains from where we came so that path is minimum.

Order of Removal

1	(0, 0)
2	(1, 10)
3	(2, 10)
4	(3, 10)
5	(4, 2)
6	(5, 3)
7	(6, 3)

vis

If shortest dist. from source to destination
 if (node == dest)
 return cost;

To get path

vector<int> path;

while (dest != -1)

{

ans.push-back(dest);

dest = par[dest];

}

reverse(path.begin(), path.end());

4) DFS → $O(E)$

→ goes deeper until no further node is nbr.

void dfs(vector<vector<int>> &adj, int v, int src)

{

~~if (vis[src] == 0) continue~~

cout << src << " "; vis[src] = 1;

for (int nbr : adj[src])

{

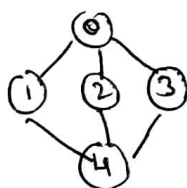
if (!vis[nbr])

dfs(adj, v, nbr);

}

}

0 1 4 2 3



5) Topological Sorting

3 COLOURING ALGORITHM → $O(E)$

→ Topological sorting

→ Check cycle in Directed graph

→ Vertices that form cycle.

bool dfs(vector<vector<int>> &adj, vector<int> &vis, int src)

{

if (vis[src] == 2) return 1; // no cycle, src has completed

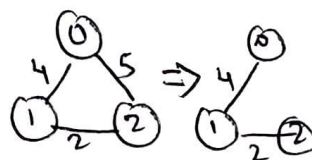
// all its calls

if (vis[src] == 1) return 0; // cycle src has not completed its calls & you visited it again

Prims

→ MST [any weight]

→ pq.push({nbr, cost of nbr});

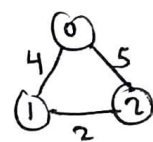


→ get shortest path

Dijkstra

→ Shortest path [+ve edge weight]

→ pq.push({nbr, cost + cost of nbr});



src: 0
dest: 2
cost: 5

src: 0
dest: 1
cost: 4

→ minimise overall weight

vis[src] = 1; // start visiting src

for (int nbr : adj[src])

{

if (dfs(adj, vis, nbr) == 0) return 0;

// cycle is present.

}

vis[src] = 2; // source has visited all

ans.push-back(src); // his child nodes w/o cycle.

return 1; // no cycle found;

}

→ pushing into my ans when all outdegrees are 0 or visited.

void toposort(vector<vector<int>> &adj, int v)

{

vector<int> vis(v, 0);

for (int i = 0; i < v; i++)

{ int n = dfs(adj, vis, i);

if (n == 0) // cycle

}

if cycle is present

for (int i = 0; i < v; i++)

{ if (vis[i] == 1) cout << i << " ";

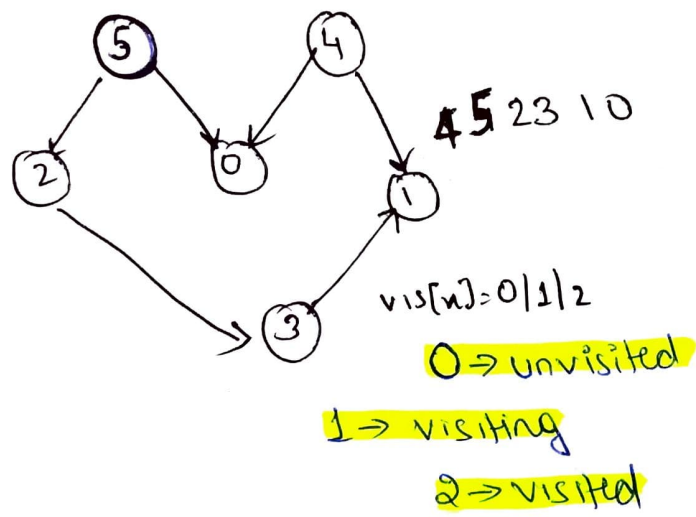
}

vertices in cycle

reverse(ans.begin(), ans.end());

for (int i : ans) cout << i << " ";

Topological Sorting



BFS (KAHN'S ALGO) - $O(E)$
 ↳ detect cycle
 ↳ Topological sorting

```

queue<int> q;
for (auto it: arr)
{
  for (int i: it) indeg[it]++;
}

for (int i=0; i<V; i++)
{
  if (indeg[i] == 0) q.push(i); // 5 4
}

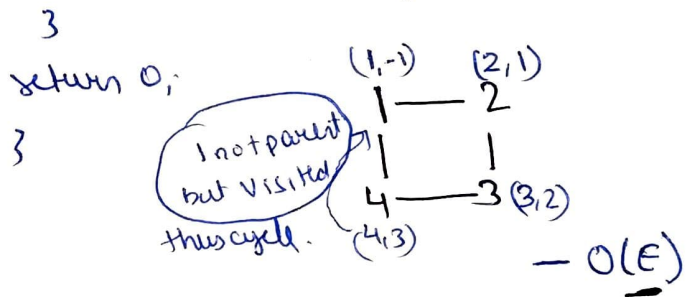
while (!q.empty())
{
  int u = q.front(); q.pop(); // 5
  ans.push_back(u);
  for (int i: arr[u])
  {
    indeg[i]--; // 0-1, 2-0
    if (indeg[i] == 0) q.push(i); // 2
  }
}
  
```

• if any indeg is non 0 → cycle is present

6) Cycle Detection in Undirected Graph

```

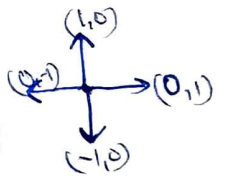
bool dfs(vector<vector<int>>& arr, vector<int> vis, int src,
        int par)
{
  vis[src] = 1;
  for (int nbr: arr[src])
  {
    if (!vis[nbr])
    {
      if (dfs(arr, vis, nbr, src)) return 1;
    }
    else if (nbr != par) return 1;
    // ie nbr is visited & not the
    // parent ie from where we came.
  }
  return 0;
}
  
```



7) Flood Fill

```

int dx[4] = {0, 0, 1, -1}
int dy[4] = {1, -1, 0, 0}
  
```



```

for (k=0; k<4; k++)
  x = i + dx[k]; y = j + dy[k];
  
```

```

int dx[8] = {0, 0, 1, -1, 1, -1, 1, -1}
int dy[8] = {1, -1, 0, 0, 1, -1, 1, -1}
  
```



8) DSU - Disjoint Set Union

- Union And

Uses → Cycle Detection [only in undirected]

→ Kruskal → Shortest Path Algo

→ Number of connected components

→ Number of nodes in a component using size

class DSU {

public:

int n; // no. of vertices, components

vector<int> parent;

vector<int> size;

DSU(int n)

{

 this->n = n;

 for (int i = 0; i < n; i++)

 {

 parent.push_back(i);

 // or parent[i] = i;

 size.push_back(1);

 // or size[i] = 1;

 }

int findparent(int n)

{

 if (parent[n] == n) return n;

 return parent[n] = findparent(parent[n]);

 // Path Compression

3

void unite(int x, int y)

{

 int px = findparent(x), py = findparent(y);

 if (px == py) return;

 if (size[py] < size[px]) swap(px, py);

 // as I want to make py parent of px

1) so it must have more nodes

 parent[px] = py;

 size[py] += size[px];

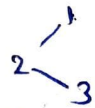
 n--; // component ↓ with every unite

3

3;

#1 Check if cycle is present

x=1, y=3



if (findparent(x) == findparent(y))

 // there would be cycle if

 // we write

else

 unite(x, y)

#2 No. of connected components

dsu.n

#3 No. of nodes in connected component

dsu.size[findparent(n)]

Static Graph

Dynamic Graph

DFS > DSU

$O(E)$ $O(E \log V)$

DFS < DSU

$O(Q * E)$ $O(Q + E \log V)$

		j →					
		0	1	2	3	4	5
i ↓	0	0	1	2	3	4	5
	1	6	7	8	9	10	11
	2	12	13	14	15	16	17
	3	18	19	20	21	22	23

$i * m + j$

$2 * 6 + 5$
 \downarrow
 no. of columns

Bonus

9) Euler Path & Circuit

Undirected & Directed [U & D]



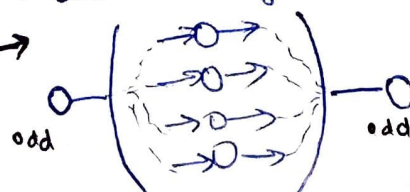
Euler Path → • visit edge only once
 (EP) • visit vertex more than once
 • start & end at diff. vertex

Euler Circuit → • same as euler
 (EC) • start & end at same vertex

UEP → only 2 vertex odd degrees

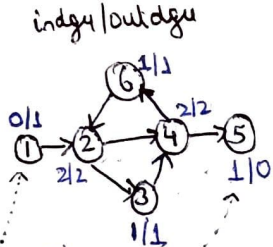
VEC → all even degrees

DEP →

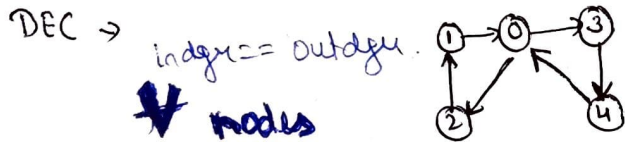


even in (indeg == outdeg)

- for all nodes
 $indeg_u == outdeg_u$
 except 2 nodes



- 1st node $\rightarrow indeg_u = outdeg_u$
- 2nd node $\rightarrow indeg_u = outdeg_u + 1$



- 10) KRUSKAL \rightarrow MST
 $\rightarrow O(E \log V)$
 \rightarrow Using DSU
 \rightarrow Greedy
 \rightarrow graph may or may not be connected

```
class DSU {
    ... (8) ...
}
```

```
int Kruskal(vector<vector<int>>& arr) {
    sort(arr.begin(), arr.end(), cmp); // sort on weights
    DSU dsu(8); // number of vertices.
```

```
    int ans = 0;
```

```
    for (auto it : arr) {
```

```
        if (findparent(it[0]) != findparent(it[1])) // check cycle
            ans += it[2]; // add weight
            unite(it[0], it[1]);
    }
    return ans;
}
```

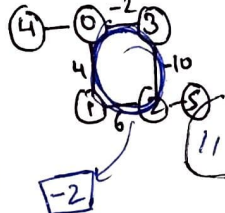
11) Bellman Ford \rightarrow Single Source shortest path
 $O(V \times E) \rightarrow [vector<int> path(V, INT_MAX);$
 $path[src] = 0;$

```
for (int j = 1; j <= V; j++) {
    for (int i = 0; i < E; i++) {
        int u = arr[i][0], v = arr[i][1], wt = arr[i][2];
        if (path[u] == INT\_MAX) continue;
        path[v] = min(path[v], path[u] + wt);
    }
}
```

So, to detect -ve wt cycle,
 at $j = V$ ie V th time

```
if (path[v] > path[u] + wt)
```

```
1 if (j == V) return 1; // cycle of -ve
   path[v] = path[u] + wt; // weight, as
                           // v-1 time min
                           // update hojao
                           // the but error
                           // baar normalise hua
                           // ie there is -ve wt cycle.
```



Dijkstra

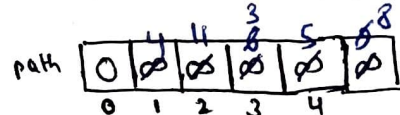
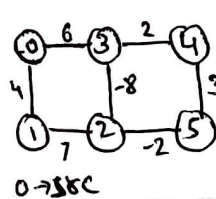
Bellman Ford

$O(E \log V)$

EV

can't handle -ve wt.
 can't detect -ve cycle
 [no -ve edge LOL]

can handle -ve wt
 can detect -ve cycle



0 \rightarrow src

0-4 shortest path

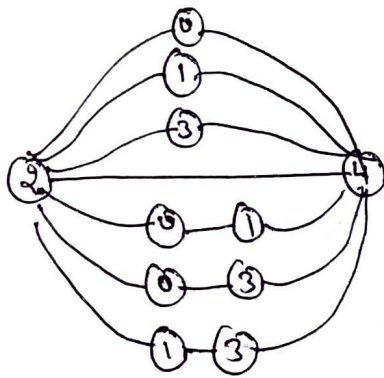
we'll get to '4' in 4 iterations [max]

- So for every vertex $(v-1)$ would be req. as path length can't be greater than $v-1$ as we can't go in cycle (coz of no -ve cycle).
- for -ve wt. finding we run loop on edges once more Δ if they are relaxed ie there is cycle, as done above.

12) Floyd Warshall

\rightarrow All pair shortest path
 $O(V^3)$
 \rightarrow adjacency matrix is helpful here

Option → Path from [2-4]



minimum we can get path from anywhere.

$$(1,2) \Rightarrow \min((1,2), (1-0) + (0-2))$$



$$(3,2) \Rightarrow \min((3,2), (3-1) + (1-2))$$



$$(3-2) \Rightarrow (3-1-0-2)$$

```
for(int k=0; k<V; k++)
```

```
{
```

```
for(int t=0; t<V; t++)
```

```
{
```

```
for(int j=0; j<V; j++)
```

```
{
```

```
if(i==j || i==k || j==k) continue;
```

```
if(arr[i][k]==0 || arr[k][j]==0) continue;
```

```
arr[i][j] = min(arr[i][j], arr[i][k] + arr[k][j]);
```

```
}
```

```
}
```

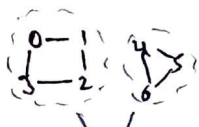
```
}
```

13) Strongly Connected Components

#Kosaraju Algorithm#

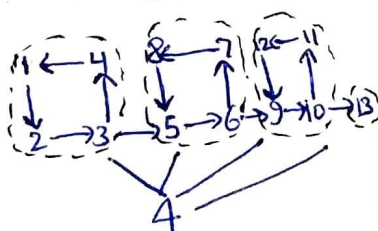
Every vertex is reachable from every other vertex.

Undirected
[Connected components]



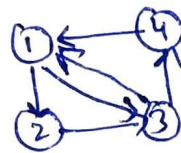
2 comp.

Directed
[Strongly Connected Comp.]



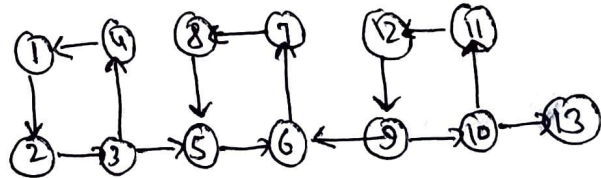
[3-5] we can't come back

Note: Count of cycle is wrong approach



2 cycles but 1 Strongly connected component.

[3-5] 5 sec wrap up isn't possible so ek specific DFS lega.



Problematic Edges

Kosaraju Algo.

Step 1: Full stack in order of departure (DFS)

Step 2: Reverse the graph

Step 3: DFS on reversed graph with on unvisited stack elements.

```
void dfs( arr, vis, src, stackcnt & s ) {
```

```
vis[src] = 1;
```

```
for(int nbr: arr[src])
```

```
{
```

```
if(vis[nbr]==0)
```

```
dfs(arr, vis, nbr, s);
```

```
}
```

```
s.push(src); // similar to topological  
// sorting, order of depart.
```

```
int kosaraju( arr, int v )
```

```
{
```

```
stackcnt > s;
```

```
vector<bool> vis(v, 0);
```

```
for(i=0->v)
```

```
{
```

```
if(vis[i]==0) dfs(arr, vis, i, s);
```

```
}
```

```
for(i=0->v) {
```

```
for(int nbr: arr[i]) rev[nbr].push_back(i);
```

// reverse graph

// to solve problem edge

```
int ans=0;
```

```
fill(vis.begin(), vis.end(), 0);
```

```
stackcnt > nouse;
```

9
10
13
11
12
1
2
3
4
5
6
7
8

s for above graph

while (!s.empty()) {

int x = s.top(); s.pop();

if (vis[x] == 0)

dfs(x, vis, x, nouse);

ans++;

}

ans = 0

dfs(9) visited
ans++; 9, 10, 11, 12

dfs(13) visited
ans++; 13

dfs(1) visited
ans++; 1, 4, 3, 2

dfs(5) visited
ans++ 5, 8, 7, 6

ans = 4

→ zeher

$O(V+E) + O(V+E) + O(V+E)$

⇒ Sort cell nodes in order of finishing time
↳ sim. to topsort

⇒ Transpose Graph

⇒ DFS acc. to finish time.

14) Mother Vertex

↳ vertices that can reach all other vertices of graph

Step 1 of Kosaraju

void dfs(—)

{

st.push(src);

}

run dfs(4)

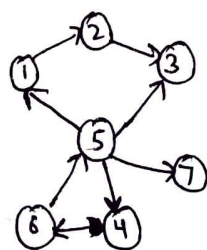
↳ if all get visited

this is answer

else no such vertex.

→ Thus return -1

Note: There can't be any answer on rest of stack as jaha nuke waha se stack bhut hai so top pe starting node hogi jisme baki tk pahuchne ki help ki



15) Articulation Point (TARJAN'S ALGO)

CUT VERTEX

↳ Undirected graph

↳ point removed disconnects graph

↳ graph divides into 2 or more comp.

if (nbr == par[src]) continue;

if (dis[nbr] != -1)

low[src] = min(low[src], dis[nbr]);

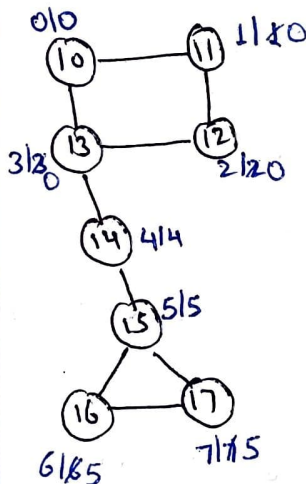
else

dfs(nbr);

low[src] = min(low[src], low[nbr]);

}

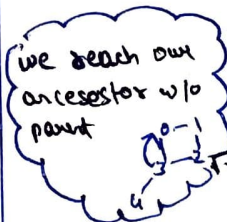
discovery / low



we need to find the order of vertices from earliest to latest to detect ~~edges~~ back edges.

∴ we use timestamp to mark nodes with increasing value ∴ we do this by assigning discovery time.

• we need to maintain earliest possible node accessible for given node which will indicate if we have back edge



Now, for source vertex ⇒ edge case

2 children

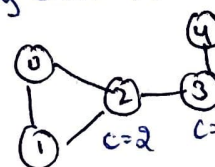


⇒ if src vertex has more than one children than if

removed divides graph into 2 or more parts, thus it's articulation point

Note: If you think for complete graph finding child > 1 would give AP X

eg.



AP: 2, 3

↳ but this is AP

so children w edge case only for source vertex.

```
vector<int> disc, low, par, ap;
```

```
void dfs(arr, int src)
```

1

```
static int time=0;
```

$$\text{dis}[\text{src}] = \text{low}[\text{src}] = \text{time} + t;$$

```
int child=0;
```

```
for(int nbr: arr[src])
```

1

```
if (nbr == par[src]) continue;
```

if (disc[nb] != -1) // visited nb

$$\text{low}[\text{src}] = \min(\text{low}[\text{src}], \text{dis}(\text{nbr})),$$

else {

child++;

$path[nbr] = src;$

```
dfs(aux, nbr);
```

$$\text{low}[\text{src}] = \min(\text{low}[\text{src}], \text{low}[\text{nbr}]),$$

if (par[src] == -1 and child > 1) // source
ap[src] = 1; // vertex

```

if (par[src] != -1 and low[nbr] >= disc[src])
    ap[src] = 1; // more nbr more bad hi
                // find hange i.e they are dependent
                // on me thus, on my removal
                // graph disconnects.

```

in main()

```
int main()
{
    par.resize(v, -1); disc.resize(v, -1); low.resize(v, -1); ap.resize(v, 0);
}
```

$$\text{arr}[u].\text{push-back}(v);$$
$$-v \quad \text{---} \quad (u)_i$$
$$d f_S(\alpha, 0);$$

$f_8(\text{int } i \geq 0; i \leq v, i + e)$

4

}

Note

#1 Shortest Path Algo \rightarrow BFS edge count

$\left\{ \begin{array}{l} \rightarrow \text{Prim} \\ \rightarrow \text{Kruskal} \end{array} \right\}$ overall graph minimise

$\left\{ \begin{array}{l} \rightarrow \text{Dijkstra (+ve edge wt.)} \\ \rightarrow \text{Bellman Ford} \\ \rightarrow \text{Floyd Warshall} \end{array} \right\}$ (ElogV) (EXV) (V³)

#2 Dickstra, Bellman Ford, Floyd
warshall all work for directed &
undirected.

→ bridge that edge jisko remove krne se we get 2 diff components.

low[mbx] > disc[src]

only charge
baki same

ie edge
may she (sac)
se bhi ~~to~~ nhi
milha chahiye
uska bad wala
time pe
mule than
it's bridge

```
op.resize(v, 0);
```

all articulation point in graph