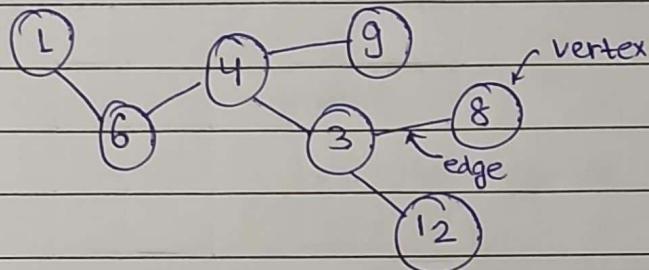


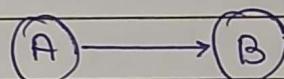
GRAPHS

→ network of nodes



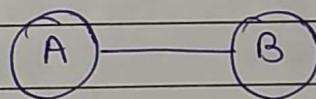
• Edges :-

✓ ① Uni-Directional / Directed Graph



(we can't go from B to A)

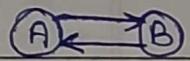
✓ ② Undirected Graph



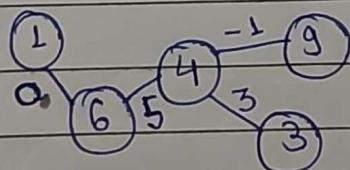
can go from B to A
or A to B

also represented

as Bidirectional
Graph

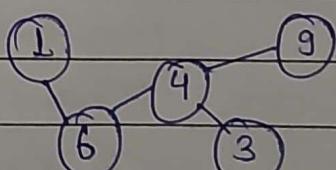


✓ Note:- every edge has a property associated with it called Weight.



$(0, 5, 3, -1) \equiv$ weights

Weighted Graph
(un-directed)



Unweighted
graph

✓ Storing a Graph

- Adjacency List
- Adjacency matrix
- Edge List
- 2D matrix (implicit)

Defining a structure / representing a graph.

Since Linked list and Binary Trees represented by making class `Node` {
 this also directly
 comes under Java}

`data,`
`left,`
`right`

Collections Framework.

⇒ I. Adjacency List

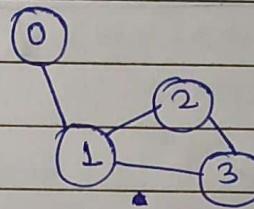
↓
 List of Lists

0 → (1) → List[0]

1 → (0, 2, 3) → List[1]

2 → (1, 3) → List[2]

3 → (1, 2) → List[3]



✓ vertices = 4 (no. of nodes)

✓ edges = 4

✓ unweighted,

✓ undirected.

can be stored as -

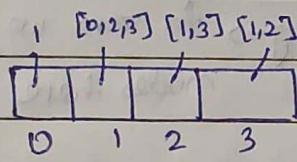
ArrayList < ArrayList >

check codes
of some
(earlier)

Array < ArrayList >

HashMap < int, lists >

↓
 vertex
(key)
value



Benefits :-

✓ extra info.
(1) optimised

✓ find your neighbour O(1) e.g. (0, 1, 2, 3) in ↗
 ↗ (ArrayList < ArrayList >)

elements

✓ ArrayList of ArrayList < Edge >

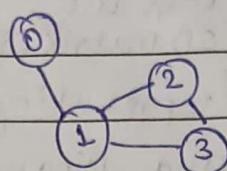
source destination weight by default (1) we're letting this by own.

✓ e.g. edge b/w 1 & 3 has → source = 1

✓ destination = 3

weight = 1 (taking as default)

⇒ II. Adjacency matrix

 $V = 4$ $E = 4$

↑ space req for making

adjacency matrix $\rightarrow O(V^2)$

more than adjacency list

•	1	2	3	
0	0	1	0	0
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0

→ You know how this is filled

problem checking one no. 2 times

once 0 with 1 in 1st row then

again 1 with 0 in 2nd row

in list we were getting

 $O(k)$ space complexity

no. of edges

↑ true complexity $O(1)$ but here $O(V)$

no. of vertices

for like this →

e.g. → for zero we're only

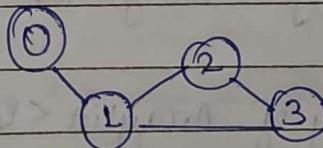
checking no. of its edges

e.g. → for zero we are checking all

nodes here.

Benefit :- we can store weighted info. also in matrix.

⇒ III. Edge list



$$\text{Edges} = \{ \{0, 1\}, \{1, 2\}, \{1, 3\}, \{2, 3\} \}$$

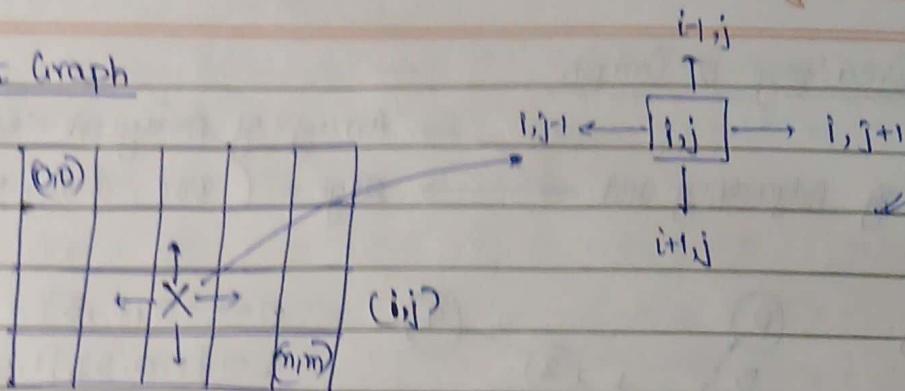
$e_1 \quad e_2 \quad e_3 \quad e_4$

also store weights → $\{ \{0, 1, 4\}, \{1, 2, 0\}, \{1, 3, -4\}, \{2, 3, 5\} \}$

$\longleftrightarrow e_1 \quad \longleftrightarrow e_2 \quad \longleftrightarrow e_3 \quad \longleftrightarrow e_4$

e.g. Min. Spanning Tree → where we sort edges using edge list
it becomes easier.

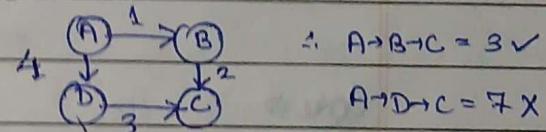
Implicit Graph



Applications of Graphs

- a. Shortest path in MAPS e.g., going from A to C (feasible?)

Dijkstra's, Bellman Ford

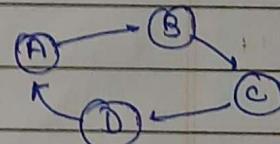


$$A \rightarrow B \rightarrow C = 3 \checkmark$$

$$A \rightarrow D \rightarrow C = 7 \times$$

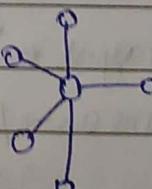
- b. Implement connection b/w profiles (SOCIAL NETWORK)

- c. Shortest cyclic route (DELIVERY NETWORK)



Shortest cyclic route to deliver at B, C, D & return.

- d. Expressing molecules (Phy. + Chem)



- e. Routing algorithms (fastest data transfer)

- f. machine learning (computation graphs)

↳ to explain neural networks for calc.

- g. Dependancy Graph

windows → vs code → extension → better code

(without windows no further process ahead like vs code)

- h. Computer Vision for Image segmentation.

detecting face, eyes, etc.

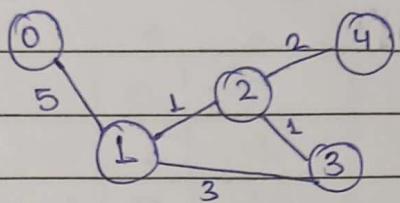
- i. Graph databases.

- j. Research (human neurons connections)

Creating a Graph

using Adjacency list → Array of ArrayList <edges>
Edge = (src, dest, weight)

e.g.



$$0 \rightarrow \{0, 1, 5\}$$

$$L \rightarrow \{L, O, S\}, \{L, R, T\}, \{L, B, D\}$$

$$2 \rightarrow \{2, 1, 1\}, \{2, 3, 1\}, \{2, 4, 1\}$$

$$3 \rightarrow \{3, 2, 1\}, \{3, 1, 3\}$$

$$4 \rightarrow \{4, 2, 2\}$$

Note:- vertex & source will always be same.

code =>

public class Edge {

```
int arr;
```

```
int dest;
```

int wt;

const^r of all

۳

P8UTM {

int v = 5; v=vertices

array of arraylist \leftrightarrow `ArrayList<Edge> [] graph = new ArrayList[V];`
↳ [same as \rightarrow `int[] arr = new arr[v];`]
↳ array of integer

initially, ArrayList if all elements are null so we've to convert it into empty ArrayList.

↳ Basically adding ArrayList to all index of array.

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

Creating empty ArrayList to all ← graph[i] = new ArrayList<>() ;
index.

✓ Adding edges info at vertices →

// 0th vertex
graph[0].add(new Edge(0, 1, 5));
// 1 vertex

" [1] " " (1, 0, 5));

! 1/2nd vertex, 3, 4.

✓ Neighbours

// 2^s neighbours :- where 2 neighbour
is stored

for (int i=0; i < graph[2].size(); i++) {

inside every arraylist ← Edge e = graph[2].get(i); //src,dest,wt.
edges are stored sout(e.dest);

because destinations are the
neighbours of vertex.

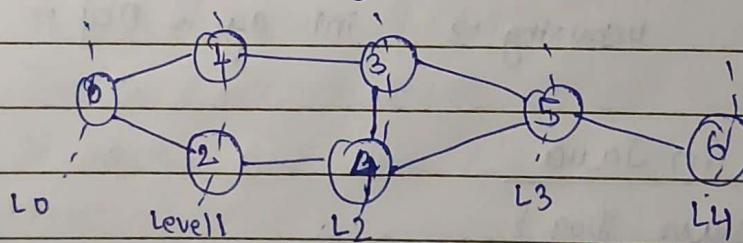
• Graph Traversals

1. BFS (Breadth First Search)

2. DFS (Depth First Search)

⇒ BFS

↳ Go to immediate neighbours first.



∴ it is also called level order traversals
(Breadth first travel karo)

↳ can't access 3 before 2 after 1.
" " 5 " 3, 4.

```
node.next = insertRec (val, index--, mode.next);
```

```
return node;
```

```
}
```

→ write before : public void insertRec (int val, int index){
head = insertRec (val, index, head);
}

BFS continued...

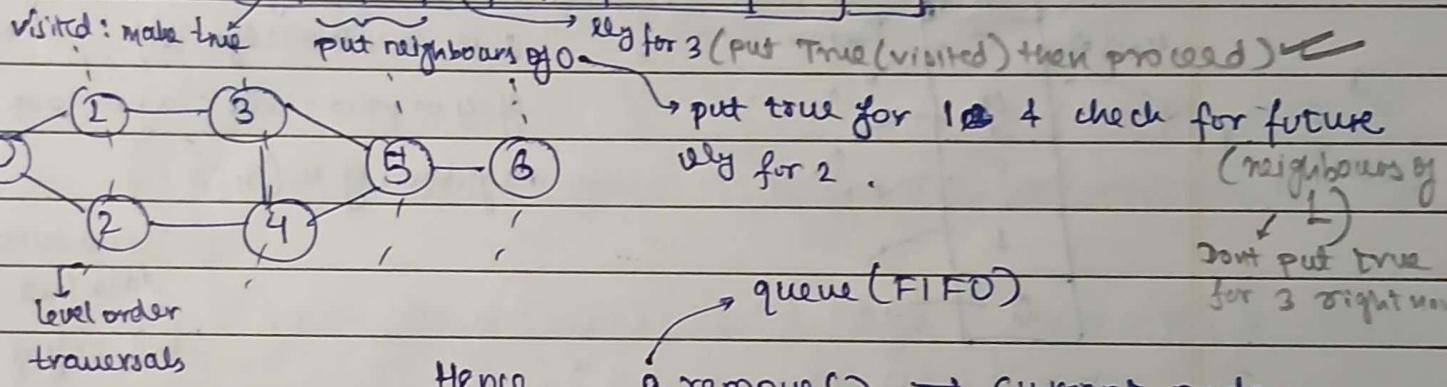
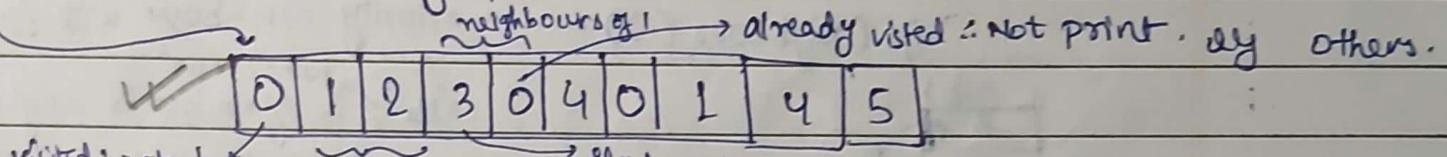
✓ Make a boolean type array and whenever we found an element make it true.

• visited :- ✓ 1. vis [current] = True

✓ 2. print it

✓ 3. neighbours add queue.

in the answer array put 1st element at 0th index



level order
traversals

Hence,

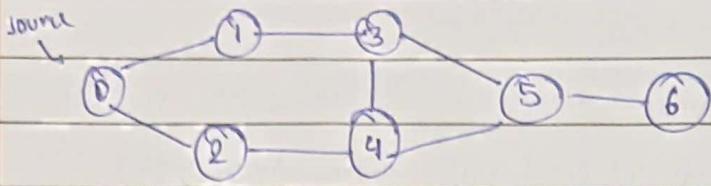
queue (FIFO)

q.remove() → current node

if (! visited[current])

visited[current] + print(current)

* DFS (Depth first search)



$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$
 $\therefore 0 \rightarrow 1(0,1) \rightarrow 3(4,1,5) \rightarrow 4(2,3,5) \rightarrow 2(0,4) \rightarrow 5(3,4) \rightarrow 6$.

→ Priority on the basis of neighbour (not on levels).

recursion (implicit stack) :- $\text{dfs}(\text{graph}, \text{curr}) \rightarrow e.\text{dest}$ } neighbour

① current → visit → i.e. $\text{print} + \text{vis}[\text{curr}] = \text{true}$.

② for (int i=0 to n) → for neighbours ($n = \text{total no. of neighbours}$)
 $\text{if} (\text{!vis}[\text{curr}])$ → not visited

$\text{dfs}(\text{neighbour}(i))$ → priority neighbour wise (socho)

calls 2. again loop for neighbours of
 \downarrow point to call

2 i.e. $(0,4) \rightarrow$ again loop for neighbours

of 4 → print + true : again loop for
 neighbours of 4 ($3,5,2$) - - -.

∴ true already
 data hua hai
 \therefore visit only if curr=false

: [print → 0134256] ⇒ agar kisi of neighbour already visited
 hai toh out from stack.

* code:- static void dfs (ArrayList<Edge> graph, int curr, boolean vis[]){

cout(curr + " ");

vis[curr] = true;

for (int i=0; i < graph[curr].size(); i++) {

Edge e = graph[curr].get(i);

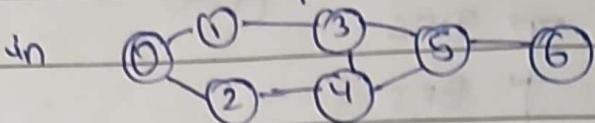
if (!vis[e.dest]) { → [e.dest] i.e. neighbour visit

y y dfs(graph, e.dest, vis);

tha yahi

Has Path?

↳ to check if 5 is also in path with 7 or 0



⇒ hasPath(graph, src, dest, vis) {

(1) src == dest

(2) ^{True} visit src

(3) if (!vis[neigh]) if hasPath(neigh, dest))

True

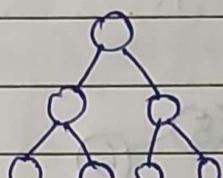
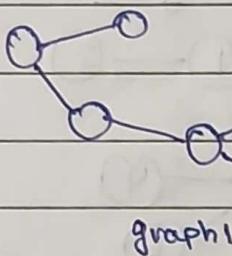
}

else false

neighbour is not
visited) and (neighbour is
e.dest) ∴ true

04/09/24

Connected Components



graph3

Combining graph 1, 2, 3 together is ^{also} a graph with multiple connected components.

If we connect them (1 & 2) via 1 edge ∴ connected component will be 1.

DFS

vis[] → boolean → initialize all with F. eg.

for (int i = 0; i < graph.length);

if (!vis[i])

y dfs util

→ using this we can

visit from 0 to 6 etc.

↪ util name is convention for helper func.

dfsutil {
} DFS

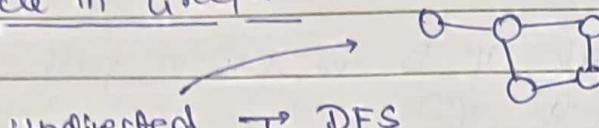
} helperfunction

you know what is an helper func.

concept of subpart for main func.
e.g. swap func. is a helper function.

↪ code for BFS as well as DFS.

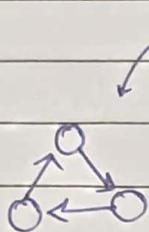
* Detect Cycle in Graph



Undirected → DFS

→ BFS

→ DSU (Disjoint Set Union)



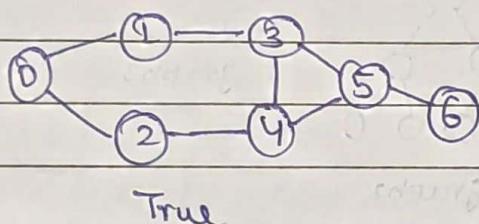
Directed → DFS

→ BFS

→ Topological Sort (Kahn's Algo.)

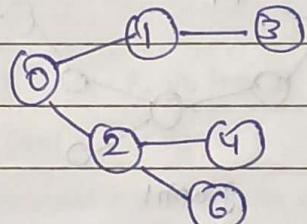
I. Undirected Graph

cycle

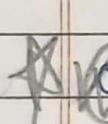


True

no cycle



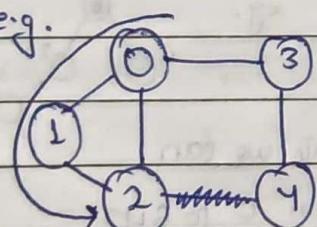
false



Condition: When the neighbour of any node is already visited.

parent is the node through which current comes.

e.g.



here 1 is the parent of 2

0 " " (" ") 1 .

∴ Case 1: $\text{vis}[\text{neigh}] \checkmark$ i.e. neighbour already visited
parent \times but not its parent

Always cycle exist

∴ True

case 2: $\text{vis}[\text{neigh}] \checkmark$ $\text{parent} \checkmark$ \downarrow not necessary ;. Continue
cycle exist

case 3: $\text{vis}[\text{neigh}] \times$ \rightarrow normal DFS \rightarrow call(visit)
if cycle \rightarrow !. True.

Since, we're starting from 0 \therefore let its parent be -1.

~~detectcycle~~(Util(graph, vis, curr, parent))

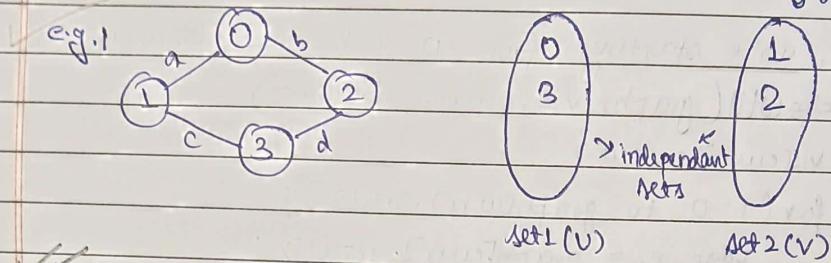
```
vis[curr] = true;
for(i=0 to graph[curr].size()){
    edge e = graph[curr].get(i);
    case 1  $\leftarrow$  if(vis[e.dest] if e.dest != parent){
        return true; } neighbour
    }
}
```

```
case 3  $\leftarrow$  else if(!vis[e.dest]) if detectcycleutil(graph, vis, e.dest, curr)
    return true; }  $\rightarrow$  soch-easy.
    { return true; } if we break neighbour
    // do nothing for case 3
    return false; }  $\rightarrow$  cycle does not exist..
```

\downarrow
Check [for removing edge still true].

* Bipartite Graphs

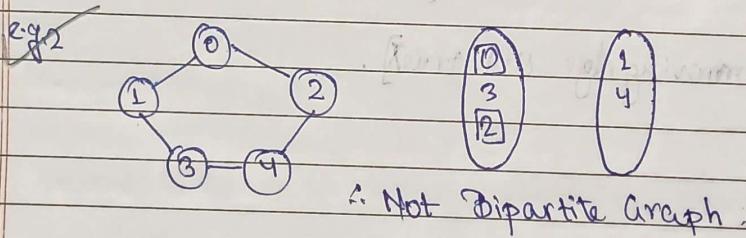
for every edge (u, v) either u belongs to U and v belongs to V or u belongs to V and v to U i.e. there is no edge that connects vertices of same set.
 $U \cup V$



Koi bhi ek aik edge wala to jisse connecting nodes ek set ki mbi hone chahiye for whole graph then that graph is called Bipartite.

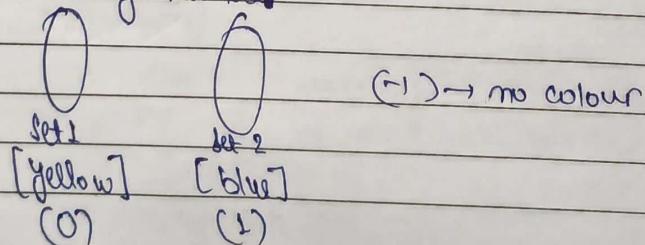
here, $c \rightarrow 1$ and $3 \downarrow$ | $b \rightarrow 0$ and $2 \downarrow$ | $a \rightarrow 1$ | $d \rightarrow 2$

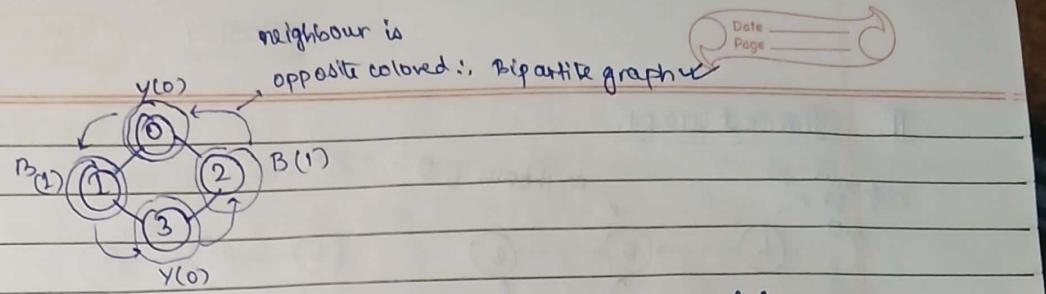
Since, this graph can be divided into two sets with condition* ∴ It is a Bipartite graph



- How to check?

⇒ Coloring method :-



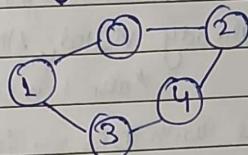


care1 : neigh \rightarrow col \rightarrow same \therefore false
 care2 : " \rightarrow " \rightarrow diff. \therefore True
 care3 : neigh \rightarrow no color \rightarrow do opp. coloring + push in Queue

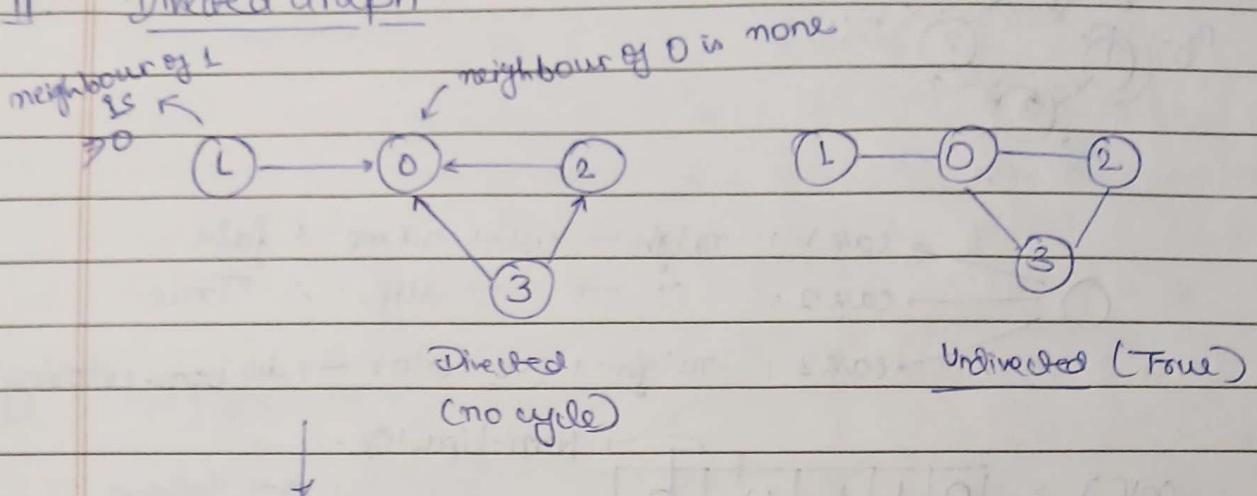
$col[i] = [0 \ 1 \ 1 \ 0 \ b]$
 Non-bipartite.
 T.C. $\rightarrow O(V+E)$.

0 \rightarrow yellow
 1 \rightarrow blue
 -1 \rightarrow none

$0 ? 1 : 0$
 ternary operator
 until if 0 then next one will be 1 & if not 0 then
 next one will be 0.



II. Directed Graph



Why undirected DFS approach fails?

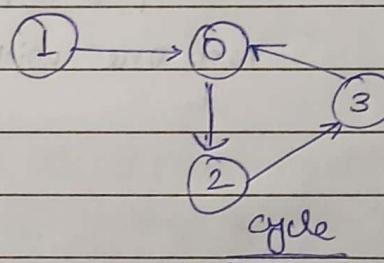
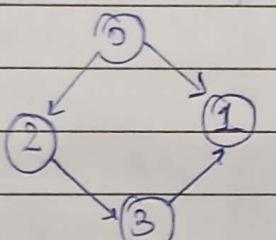
Here → parent of 1 is -1 & neighbour is 0

Since 0 is already visited, still no cycle

point A. *also 1 is not parent of 0*

Hence, parent with DFS → fails → [neighbour → vis → not parent]

e.g.



Approach:-

We know the basic pseudocode for DFS:

```
dfs(current){
```

```
    vis[current] = true;
```

```
    for (all neighbours){
```

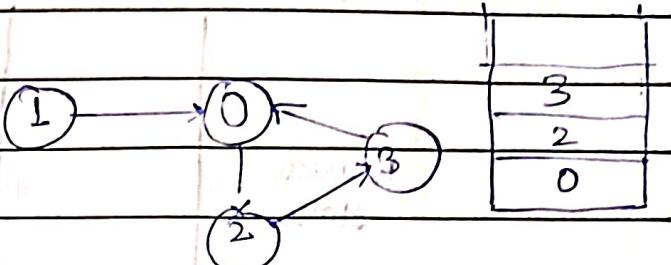
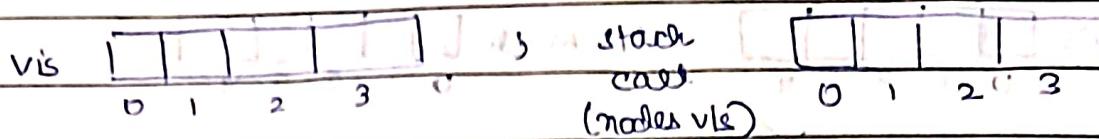
```
        if (!vis[neigh])
```

```
            visit ✓
```

```
}
```

modification :-

We'll make a stack which tracks the record the current neighbours unlike boolean visited array.



Modified Pseudocode :-

dfs (curr){

 vis[curr] = true;

 stack[curr] = true; \rightarrow making this also true to show graph.length. we call it under aa chuka hai

 for (all neighbours){

 if (stack[neigh] = true){ \rightarrow to check kisi pehle hi aag stack aa gya hai.

\therefore cycle exist \rightarrow True

(here o pehle hi stack aag chuka hai for 3).

 if (!

 if (!vis[neigh]) \rightarrow if neighbour is not visited

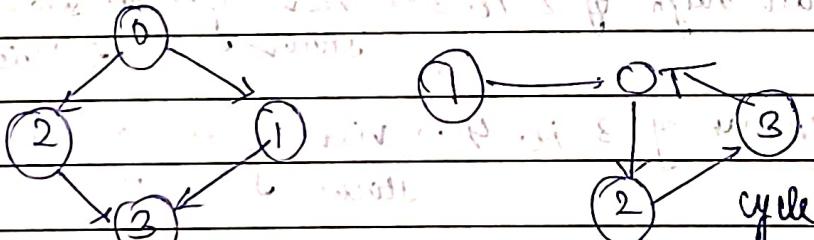
 isCycle(neigh) \rightarrow True \rightarrow if not visited then make it True.

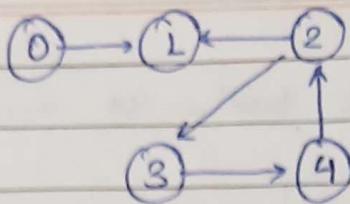
 stack[curr] = false

\rightarrow stack aag upar upar jaate waqt jab true krya tha waqt hi niche (pop)

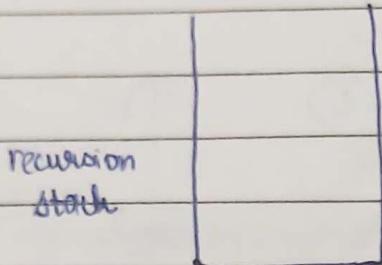
 stack aag upar upar jaate waqt jab true krya tha waqt hi niche (pop)

Dry Run karke dekho for :- (or refer video)



Example →

* vis [] ; stack []
 0 1 2 3 4 0 1 2 3 4



1. Start from 0 → vis ✓ } true
 stack ✓

2. 0 → checks ki stack ki neighbour exist karta hai ya nahi? → No
 (∴ saare neighbour
abhi unvisited
hai.)
 Therefore visit kardo.

3. 1 → vis ✓ } true
 stack ✓

4. NO neighbour of 1 → return to 0 → 1 removes from stack
 ∴ 1 → false → no neighbour now of 0 also ∴ 0 → false.

5. Now start from 2 → vis ✓
 stack ✓

6. check for neighbour of 2 i.e. 1 → already visited! but not in
 STACK → ∴ no importance of 1 → don't take any action on 1.

7. check for next neigh. of 2 i.e. 3 → vis ✓ } ∵ unvisited then
 stack ✓ i.e. !vis[e.dest]

8. " " neighbour of 3 i.e. 4 → vis ✓ } " " "
 stack ✓ " "

9. " " " " 4 i.e. 2 → '2' already exists in the stack
 Hence, cycle exists.

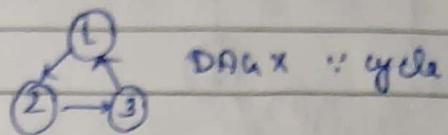
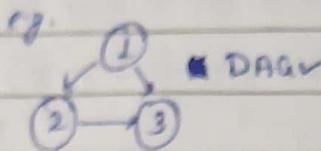
Time Complexity: $O(V+E)$
 ↳ DFS.

extra read: Stack Overflow → why DFS and not BFS for cycle detection in graphs?

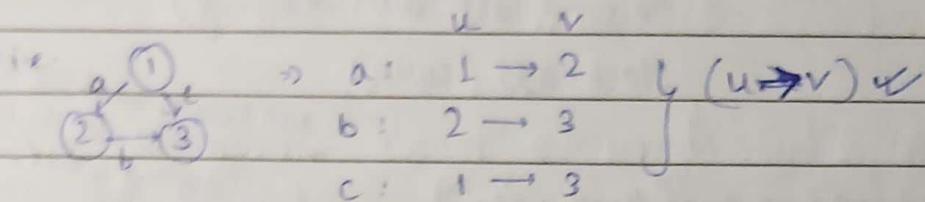


* Topological sorting

→ works for DAG (Directed Acyclic graphs) only.



→ It is a linear order of vertices such that every directed edge $u \rightarrow v$, the vertex u comes before v in order.



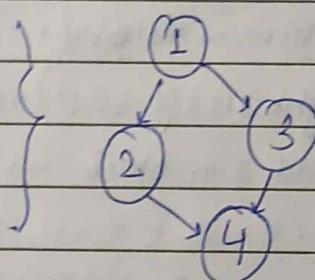
* Dependency * → Konsa kaam kab perform hoga since konsa kaam kispe depend karta & → mtlb konsa kaam kis kaam ah bina nhi ho skta.

eg. You cannot serve maggie before boiling water.

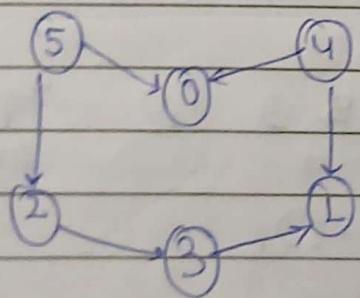
eg. You " deploy a slw before fixing bugs

Action 1: boil water

- " 2: add maggie
- " 3: add masala
- " 4: serve maggie.



Example:-



possible runs:

5 4 2 3 1 0

4 5 2 3 1 0 → only after 4 and 5.

Topmost

formore

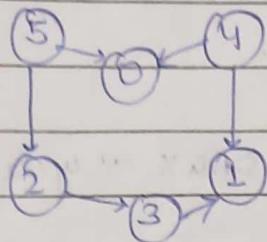
2 will come after

5 only seen once for

3 (only after 2)

Approach → modified DFS

"track what comes first"



DFS → stack add from stack is a LIFO

∴ Jo last h' aaya hai wahi sabse

pethe niklega (e.g. 5)

∴ put 5 at last.

dfs(graph, curr, vis, stack) {

vis[curr] = true;

neigh → call (unvisited only)

stack.add(curr)

→ add here as based yds stack it
will give me topological
sorting order.

for (int i=0; i < graph.length; i++) {

if (unvisited)

dfs()

} } to visit all vertex
(connected components)

Dry run:-

1. Add 0 → vis✓ no neighbour → stack.add ✓

2. ∵ 1 → vis✓ " " → " " ✓

3. " 2 → vis✓ → neighbour of 3 (unvisited) ∴ call → add 3 in stack

check neighbour of 3: 1 (already visited) → no other neighbour

of 3 → add 3 in stack → no other neighbour of 2 → add 2

in stack.

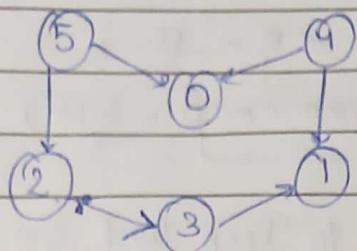
4. Add 4 → vis✓ → no unvisited neighbour → stack.add ✓

5. " 5 → " " " "

When stack pops out ⇒ 5 4 2 3 1 0 → Topological sorting order

Time Complexity: $O(V+E)$.

* Topological Sort using BFS "Kahn's Algorithm"



	0	1	2	3	4	5
in-degree:	2	2	2	1	0	0
(incoming)						

	0	1	2	3	4	5
out-degree:	0	0	1	1	2	2
(out-going)						

* Fact: A DAG has at least 1 vertex with in-degree 0 and one (Observation) vertex with out-degree 0.

* We know Topological ^{sort} works on dependency, so we can conclude that the top most node has no dependency with in-degree = 0.

indeg:	2	2	1	1	0	0
	0	1	2	3	4	5

BFS (loop on Queue)

1. Put first those nodes with in-degree 0. ∵ Queue is FIFO.

~~1 | 5~~ → topological sort logic

point 4 → then check for its neighbours

if in-degree → 0 ∴ add in queue

∴ Decrease in-degree by 1 for 0 and 1.

⁽²⁾ ↓ ⁽²⁾ ↓

2. Now print 5 → then check for its neighbour → decrease their in-degree

by 1 ⁰ ↓ ² ↓ → if both got in-degree zero → add in Queue.
¹ ↓ ⁰ ↓

~~1 | 5 | 0 | 2 | 3~~
print 0

3. 0 has no neighbour ∴ no change

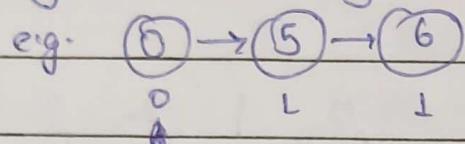
4. 2's neighbour → 3 → in-degree reduced by 1 ∴ now become zero ∴ add in Queue.

Print
~~1 | 5 | 0 | 2 | 3~~

5. Print 3 → 3's neighbour: 1 → in-degree(1) → 0 → add in Queue: ~~1 | 5 | 0 | 2 | 3 | 1~~

6. No neighbour of 1 ∴ no change
Print 1. ~~1 | 5 | 0 | 2 | 3 | 1~~ Hence I/O/P = 4 5 0 2 3 |

• Why no need of visited array check?



∴ print 0 → reduce neighbor by 1 → 5 → Only 6

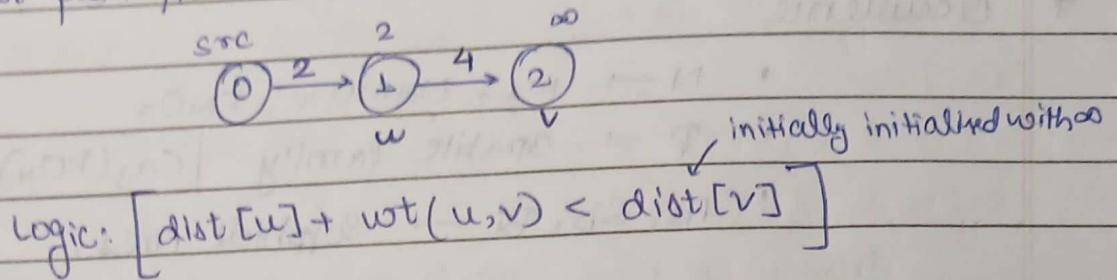
now become O(m-degree)

Matlab see yehle sabhi \rightarrow indegree

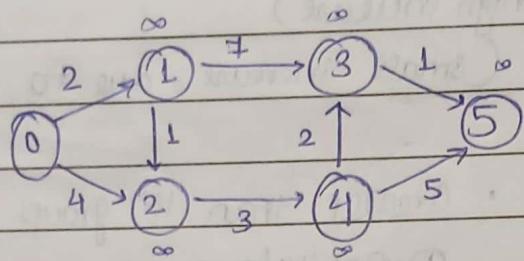
O hi honge for sure ∴ no need to keep track via visited array.

* Dijkstra's Algorithm

- greedy based algorithm.
- Shortest path from source to all vertices (weighted graph)



$$\therefore \text{dist}[v] = \text{dist}[u] + \text{wt}(u,v)$$



Source=0

BFS manner

- update all neighbours of source (hamesha direct distance update hota hai) $1 \rightarrow \infty = 2$
 $2 \rightarrow \infty = 4$

- Now check for not visited + shortest $\therefore 1$.

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

$\downarrow 2 \rightarrow 4$

- 1st neighbour of 1 $\rightarrow 3 (\infty) \therefore \text{dist}[4] + \text{wt}(u,v) < \infty = \text{dist}[v]$

$$\therefore \text{dist}[v] = \text{dist}[1] + 7 = 2 + 7 = 9$$

My 2nd neighbour of 1 $\rightarrow 2 (4) \therefore 2 + 1 < 4$

$$\therefore \text{dist}[2] = 3$$

- Again check for visit $\xrightarrow{\text{1}} (2, 3, 4, 5) \xrightarrow{\text{2}} \downarrow \therefore \text{visit } 2$.
shortest $\sqrt{5} \quad \downarrow 3$

- neighbour of 2 $\rightarrow 4 (\infty) \therefore 3 + 3 < \infty \therefore \text{dist}[4] = 6$.

- shortest $\rightarrow (4) \quad \therefore \text{visit } 4$
visited $\xrightarrow{\text{3, 4, 5}}$

7. Neighbour of $4 \rightarrow 3 \therefore 6+2 < 9 \therefore \text{dist}[3] = 8$
 " " $4 \rightarrow 5 \therefore 6+5 < \infty \therefore \text{dist}[5] = 11$

8. shortest $\rightarrow (3) \checkmark \rightarrow \text{visit} \cdot$
 9. Neighbour of $3 \rightarrow 5 \therefore 8+1 < 11 \therefore \text{dist}[5] = 9 \text{ } \textcircled{2}$

10. No Node after 5 $\rightarrow \therefore \text{visit } 5$

Hence final distance :-

$0 \rightarrow 0 (0)$

$0 \rightarrow 1 (2)$

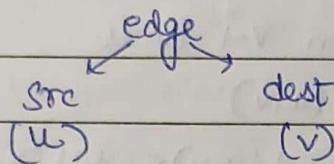
$0 \rightarrow 2 (3)$

$0 \rightarrow 3 (8)$

$0 \rightarrow 4 (6)$

$0 \rightarrow 5 (9)$

Implementation :-



① Initialise dist. Arc $\rightarrow \text{src} = 0$, else ∞

② priority Queue to store pairs (node, dist)

read pdf of greedy. $\left\{ \begin{array}{l} \text{covered in greedy} \\ \text{for shortest dist. node} \end{array} \right\}$ help of comparable comparator

while (priority queue is not empty) {

take out curr (if not visited) $\rightarrow \text{visit}$

if ($\text{vis}[\text{curr}] = \text{false}$) {

neighbours $\rightarrow \text{dist}[u] + \text{wt}(u,v) < \text{dist}[v]$

" update dist [v]

y

with priority queue $\rightarrow T.C. = O(V + E \log V)$; without PQ $\rightarrow T.C. = O(V^2)$

Check code seriously!!!!!!

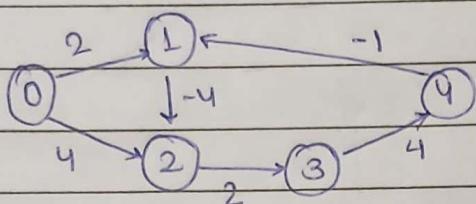
internal sorting in PQ.

dist. boolean

for visited

* Bellman Ford Algorithm

- For the cases in which negative weighted edges comes, we don't solve with Dijkstra's algo. due to any anomalies.
- DP based algorithm
- Bellman Ford can do same operation as Dijkstra + for negative edges also;
- Shortest paths from source to all vertices (negative edges)
 Time complexity $\rightarrow O(VE)$ i.e. $> T.C$ of Dijkstra
 \therefore Dijkstra is preferred for +ve edges.



$$E(\text{edges}) = 6$$

$$V(\text{vertex}) = 5$$

Perform this operation $V-1$ times
 for all edges (u, v)

if ($\text{dist}[u] + \text{wt}(u, v) < \text{dist}[v]$) { } called as
 $\text{dist}[v] = \text{dist}[u] + \text{wt}(u, v)$ } Relaxation
 } (\because we're relaxing our
 distance)

\therefore for (int $i = 0$ to $V-1$) {
 edge $(u \rightarrow v)$ } Pseudo
 relaxation } code

	0	1	2	3	4
$\text{dist}[] \rightarrow$	0	∞	∞	∞	∞
Vertex-1	1	0	2	4	-2
	2	0	2	-2	0
	3	0	2	-2	0
	4	0	2	-2	0

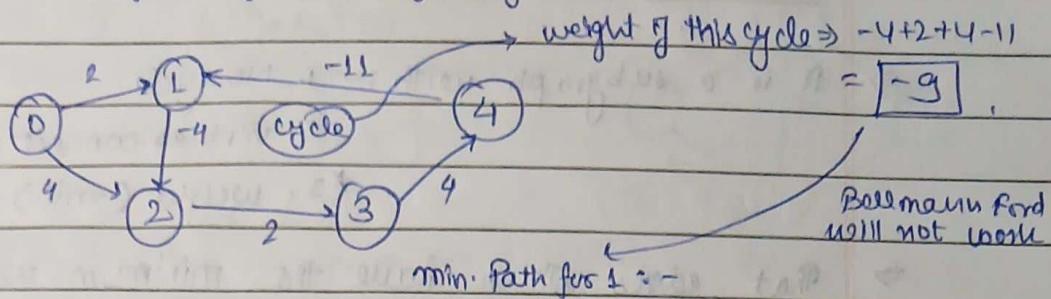
\Rightarrow ans.

Obviously it's all 4 iterations #
 same as you but sab sth hi aayenge
 always $(V-1)$ waala hi ans. hoga.

* Note: - Doesn't work for negative weight cycles.



Reasons: →



min. Path for 1: -

$0 \rightarrow 1 \dots 2$

$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \dots -7$

This will be the shortest path but we've already walked on 1 so it won't make any sense of walking over it again to get min. Path LOL.

Pseudo Code → (1). Initialise all distance except source $\rightarrow \infty$.

(2). int v = graph.length

for (i to v-1) {

for (j to graph.length) { → to get all vertices from graph

for (k to graph[j].size()) { → edges respective to all these vertex

Edge e = graph[j].get(k); → each edge \rightarrow vertex's edge

int u = e.src;

int v = e.dest;

int wt = e.wt;

if (relaxation condition) {

In Java if we add any integer

with $\infty \rightarrow$ it will make it true

if ($dist[u] \neq Integer.MAX_VALUE \text{ and } dist[u] + wt < dist[v]$)

$dist[v] = dist[u] + wt;$

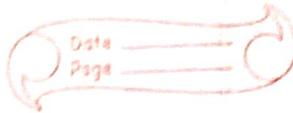
printf →

for (int i = 0 to dist.length) {

cout(dist[i] + " ");

cout();

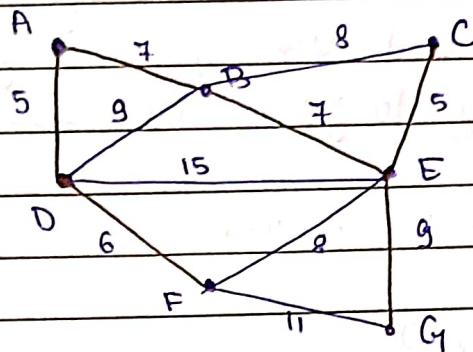
10/03/24



* Minimum Spanning Tree.

- ⇒ It is a subgraph with :-
 1. No cycle
 2. vertices connect (all)
 *3. weight (min)

⇒ That edges which have the minimum weight across all vertex which when connecting (considering only those) give a subgraph is called min. spanning tree.

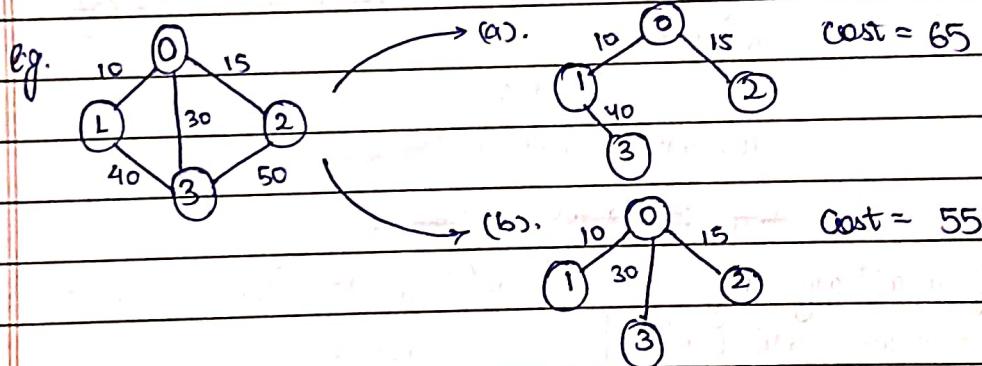


Only those vertices connected via black edges are called Min. Spanning Tree.

algorithm for solving
MST is called?

• Prim's Algorithm

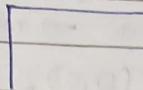
↳ works for undirected + weighted MST



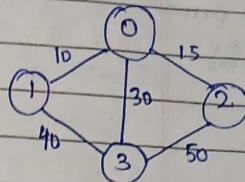
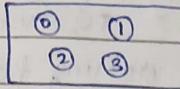
∴ cost (a) > cost (b) ∴ MST → (b) ✓
 from 'a' and 'b'.

• Dry run :-

MST set



Not in MST set



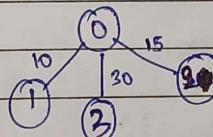
vis [] [] []
0 1 2 3

cost = 0

make it true in visit

- (1). Choose any vertex $\xrightarrow{\text{let}} 0 \rightarrow$ put in MST set & remove from not in MST set.
- (2). Make the distance of $0 \rightarrow$ Integer-Min and check for other vertex $(1, 2, 3) \rightarrow$ check their costs $0-1(10)$, $0-3(30)$, $0-2(15)$.
- (3). Select that node which have min-cost ($\text{MST} \rightarrow \text{Non-MST set}$) i.e. $1 \cdot (10)$
 ∵ Add 1 in MST set. also make cost = $0+10=10$
 \hookrightarrow make it True in visit
- (4). Check for edges going from $(0, 1)$ to $(2, 3)$ and select for minimum one $\Rightarrow (1-2) \text{ None}, (1-3) 40, (0-2) 15, (0-3) 30$.
 \hookrightarrow Min.
- (5). Add 2 (visit \checkmark) in MST set, and make cost = $10+15=25$.
 \hookrightarrow obviously remove from nonMST set with min-cost
- (6). Check for edges going from $0, 2, 1$ to 3, i.e. $(0-3) 30$,
 $(2-3) 50, (1-3) 40$.
 \hookrightarrow min.
- (7). Add edge 3 from $(0-3)$ (visit \checkmark) in MST set + cost = $25+30=55$.

Hence,



H.W Note: If in Ques. asked to store Edge \therefore store edge in MST via
ArrayList ($A[<edge>] \rightarrow$ last), Total mst cost

Prim's algo may
vary on Ques.

Don't memorize
for one purpose
only ...

Date _____
Page _____

Pseudo code \Rightarrow

1. make $vis[]$
2. " priority queue of pair type

$PQ <vertex, cost> \rightarrow$ min. pair (cost)

3. starting from $PQ(0,0)$, initialise cost=0.
4. while (PQ is not empty) {
 curr $\rightarrow (v, cost)$
 if (v not vis) {
 visit v
 add in mst (ans) \rightarrow cost
 add \rightarrow neighbours \rightarrow in PQ
 to check neighbour's edge
 can be added in PQ or not.
 }

NOTE:- A priority queue is basically a min. heap structure which sorts values, here in order to sort on the basis of vertex or cost we use implements Comparable interface $<Pair>$
 \therefore @Override

return $this.cost - p2.cost$; \rightarrow sorting on the basis
if we reverse this we'll get sort in
descending order.

also reading material \leftarrow revise Greedy (already covered)
given there. also watch later YT on comparators in Java.

* Disjoint Set [Data Structure]

Find : which set does the element belongs to

Union : merge

$m = 8$

grp1 (set1) : (1) (2) (3) (4)

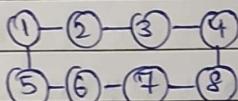
grp2 (set2) : (5) (6) (7) (8)

find(4) → set1

find(6) → set2

union(4, 6) →

set1 set2



find(6) → set1

union(1, 5) → "Cycle Detection"

↓

when we do union of elements in same set cycle is formed.

∴ It is used for : (1) Cycle Detection

(2) Konskal's algo (to find MST) → apart from Prim's

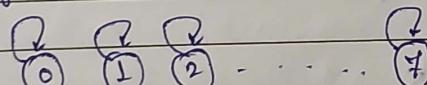
algo.

11/09/24

• Implementation :

starting → parent [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7] initially each element group to leader ↗

rank → [0 | 0 | 0 | 0 | 0 | 0 | 0 | 0]
(height) 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7



find → leader

Union → 2 groups join

For given set of instructions check how'll this works :-

next page

Union(1,3)

find(3)

Union(2,4)

Union(3,6)

Union(1,4)

find(3)

union(1,5)

(D). union(1,3)

①

• Now rank of 1 will be

increased from $0 \rightarrow 1$

• Also parent of 3 updated from 3 to 1.

(2) find(3) → • check for parent of 3
i.e. 1

• Now check rank of 1 i.e. 1

Hence find(3) → gives 1.

∴ Group leader = 1.

(3). union(2,4)

• Now 2 ya 4 any one can be leader ∴ let 2.

• Therefore leader parent of 4 \Rightarrow 2 and parent of 2 already
2.

• Also rank of 2 updated to 1.

(4). union(3,6)

• 3 का union नहीं matlab 3 का poora group का union होगा.

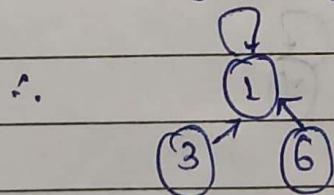
• look for rank of 6 $\rightarrow 0$, rank of 3 \rightarrow means iske parent
ki rank dekho i.e. $1 \rightarrow 1$

(group का leader की rank dekhi jaati hai sirf)

∴ 6 का group का leader 6 hi hai ∴ uska rank = 0

and 3 " " " 1 hai ∴ uska rank = 1

Hence, Iska rank kam hogा wo aake ^{bade wale ice} leader
ke saath join hogा.



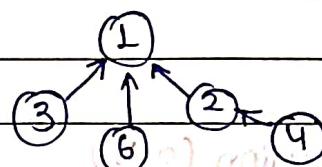
(5). union(1,4)

- Since union sirf parent mili hota hai $\therefore 1$ and 2
- Also since both have same rank \therefore koi bhi kisi se bhi join ho saka hoi. (by default $2 \rightarrow 1$ ~~not~~ join ho gaya)
- \therefore parent of 2 (now) = 1

Hence,

- rank of $1 = 2$

updated Tree \Rightarrow



- rank of $2 = 1$

- parent of $2=1$, parent of $3=1$, parent of $6=1$, parent of $4=2$

Final Answer

Note:- Final leader wo hо hoi jiska leader wo khud hota hoi.

(6). find(3)

- \therefore leader of $3?$ \Rightarrow from parent check for 3 i.e. 1 .

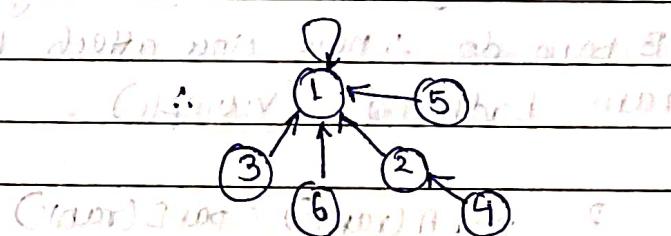
Now check parent of 1 i.e. 1 .

Hence, final leader = 1 .

eg. find(4) gives $1 \rightarrow$ socho step by step.

(7). union(1,5)

- parent of $5=1$



Final answer is 1 (after doing 5).

Ques = Find(1) ans = ?

Ques = find(1) ans = ?

Ans = 1 (because 1 is root)

- PseudoCode \Rightarrow
 - make int n, par[n], rank[n]
 - initialise $\text{par}[i] = i$ \because sab khud hi parent at first and all rank at starting $\rightarrow 0$ i.e. $\text{rank}[n] \rightarrow 0$
 - find(x)
 - \therefore for supreme parent \rightarrow if ($x = \text{par}[x]$) {
 - \downarrow return x;
 - else {
 - return find($\text{par}[x]$)

• union(a, b)

\because we find union for parent only

\therefore get parA , parB

\swarrow to get parA

\searrow use $\text{find}(a)$

\downarrow we $\text{find}(b)$ do

get parent of b

Now \Rightarrow (check for rank)

1. if equal \Rightarrow $\therefore \text{par}[\text{parA}] = \text{parB}$ if ya

\swarrow rank [parB] ++ \downarrow opposite

Jab dono rank equal ho to kisibhi ek ko (here \rightarrow parB) \Rightarrow ait

\spadesuit bada samajh do and parent of (parent of A) ait

B bana do \therefore Now since attach ho-gya to B \Rightarrow

rank badha do (v. simple)

2. $\text{parA}(\text{rank}) < \text{parB}(\text{rank})$

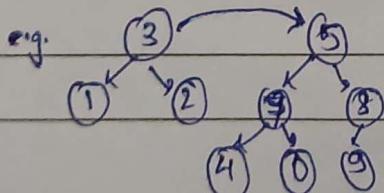
(\because same operation that $A \rightarrow B$ \Rightarrow judgega)

$\therefore \text{par}[\text{parA}] = \text{parB}$

\swarrow but rank will not change (obviously)

\therefore badhi height of tree pe chota height of tree \Rightarrow

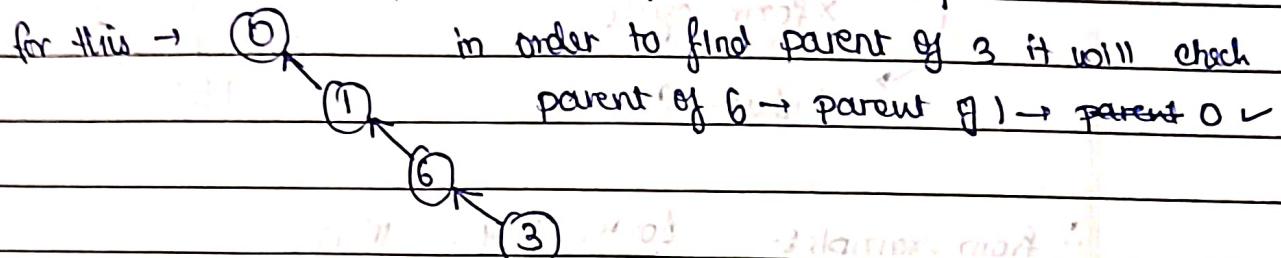
connect hone \Rightarrow off farak nhi paella.



B. if $\text{parA}(\text{rank}) > \text{parB}(\text{rank})$ then
 $\therefore \text{par}[\text{parB}] = \text{parA}$;

// check code :-

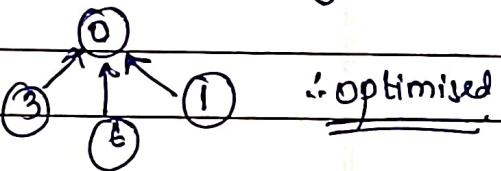
- path compression optimisation for union and find :-



rather store values for next time & keep on updating.

i.e. 3 in parent 0 stored

6 " " " "
1 " " " "



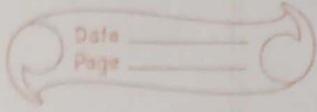
if parent updated sab se ek baar hi change ho jayega.

Note:- Time complexity of union & find comes (mathematically) as $O(4k) \rightarrow \therefore O(4) \rightarrow O(1) \rightarrow$ hence, constant time

* Therefore it is used for (1) detecting cycles in graph

(2) finding MST in Kruskal Algo.

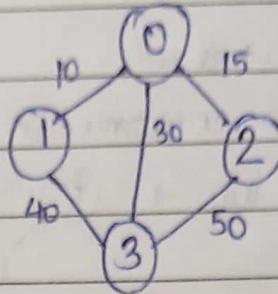
11/09/24



* Kruskal's Algorithm → for mST (greedy)

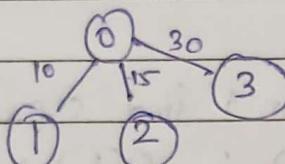
1. sort edges
2. Take min. cost edge

↓
x form cycle
include ans.



∴ from example:- Sorted Edges Weight

(0,1)	10	→ ①
(0,2)	15	→ ②
(0,3)	30	→ ③
(1,3)	40	→ X ∵ forms CYCLE
(2,3)	50	→ X ∵ CYCLE



AL<Edge> → collections.sort()

↓
gives sorting of the elements

define parameter

cost → to sort on the basis of cost

(Count MST)

for (int i = 0 to V-1)

Σ ∵ 2 vertex \nexists connect same as cycle

V-1 edges are required

- do check cycle wrt concept of union & find

- do not include wrt union (a,b) \Rightarrow $\text{par}(a) \rightarrow \text{par}(b)$

- But if parA and parB are same ∵ CYCLE condition.

- do not include in this case.

Hence →
(pseudocode)

```

for (Count = 0 to V-1) {
    Edge e → a(src)
    b(dest)
    using find(e.src) , find(e.dest)
    go to par(A) , par(B)
    if same → don't include : cycle
    if diff. → include → union(parA, parB)
}

```

c. Time complexity: $O(V + E \log E)$

// check code