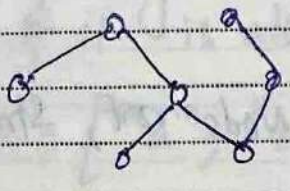
05 October
Wednesday
41st Wk • 278-087

2022 OCTOBER

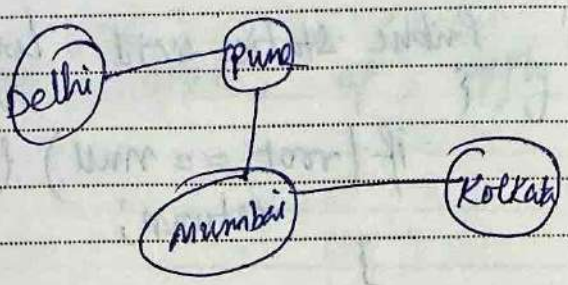
M	T	W	T	F	S	S	M	T	W	T	F	S	S
					1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31						

Graphs

It is a network of nodes



E.g → a network of cities

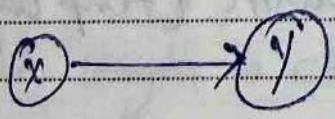


- vertex → Every node in the graph is called vertex. Multiple vertex are called vertices.
- edge → connecting lines between two vertex.

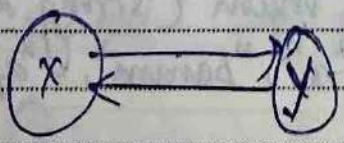
Application of Graphs

✓ Maps ✓ Social Network ✓ Delivery network

Edge can be uni-directional, or bi-directional/unidirectional

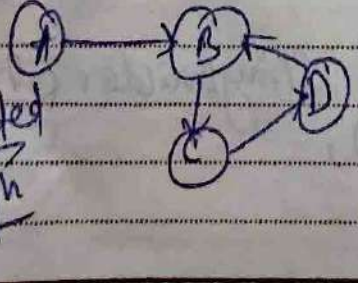


unidirectional

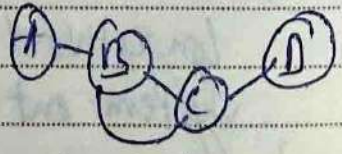


bi-directional

based on edges, graphs are Sometimes named.

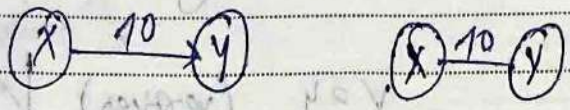


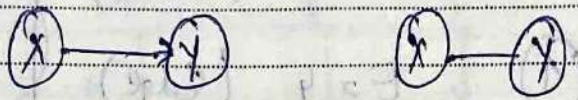
directed graph



undirected graph

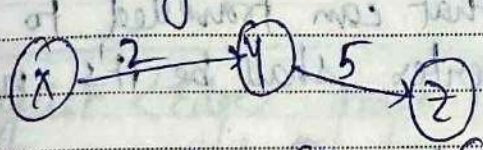
Edges based on weight

weighted 

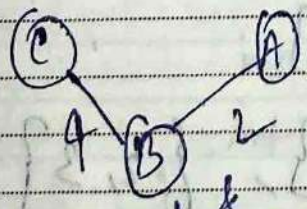
unweighted 

weight can be based on time or distance or some other quantity. For example, it can be like to travel from (X) to (Y), it takes 10 km (so, weights based on dist.)

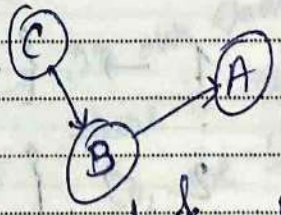
weight can also be based on time.



So, to travel from (X) to (Y), it took 2 hours, whereas to travel from (Y) to (Z), it took 5 hours.



undirected & weighted graph



directed & unweighted graph

Storing a Graph

- ✓ Adjacency list
- ✓ Adjacency Matrix
- ✓ Edge List

- ✓ 2D Matrix (Implicit Graph)

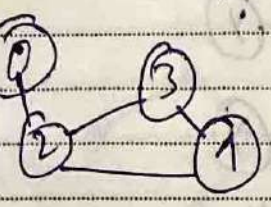
07 October
41st Wk • 280-085
Friday

2022 OCTOBER

M	T	W	T	F	S	S	M	T	W	T	F	S	S
					1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31						

Adjacency List → list of lists.

Example 1



V = 4 (vertices)
E = 4 (Edge)

undirected & unweighted.

Size of the graph = no. of vertices = 4. we shall store the information in the form of vertices.

• Neighbour → The vertex that can travelled to from a particular vertex shall be it's neighbours.

So, in this example, from (0), we can go to (2). So, 0 & 2 are neighbours.

0 → {0, 2} (source, destination)

1 → {1, 2}, {1, 3}

2 → {2, 0}, {2, 1}, {2, 3}

3 → {3, 1}, {3, 2}

So, that's how we calculate all the vertex-wise info. All these info are stored in the form of adjacency list.

There can be several ways to implement the adjacency list. Some are as follows –

(a) Arraylist of Arraylist

(b) Hashmap

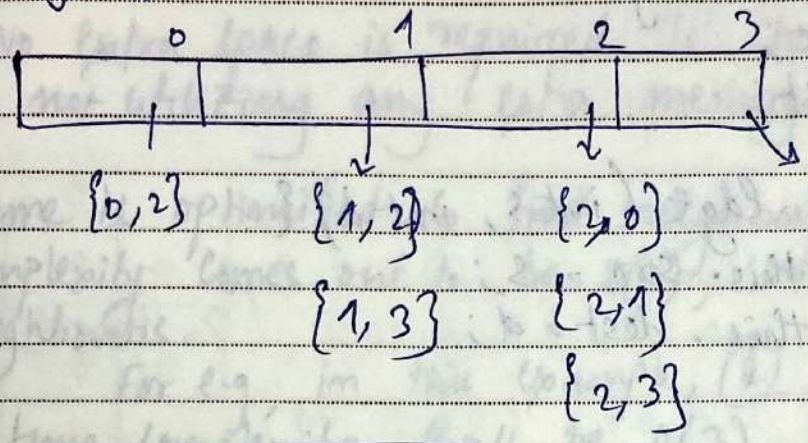
(c) Array of Arraylist, etc.

NOVEMBER 2022

October 08 Saturday

41st Wk • 281-084

Here, we shall to implement it in the manner of "Array of Arraylist".



Arraylist<Edge> graph[V]

- type of the array
- array name
- size of the array

```
static class Edge {
    int src;
    int dst;
}
```

→ In our example, there is no weight. So, we shall make a class and store src and dst.

So, to add the edges in each of the array index,

```
graph[0].add(new Edge(0,2));
graph[1].add(new Edge(1,2));
graph[1].add(new Edge(1,3));
```

09 October
41st Wk • 282-083
Sunday

2022 OCTOBER

M	T	W	T	F	S	S	M	T	W	T	F	S	S
					1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31						

```
import java.util.*;
→ Public class Classroom {
8   Static class Edge {
    int src;
9   int dest;

10  Public Edge (int s, int d) {
        this.src = s;
11      this.dest = d;
    }
12  }
    Public Static void createGraph (ArrayList<Edge> graph[])
    {
1       for(int i=0; i<graph.length; i++)
        { // create empty arraylist to store edge data
2           graph[i] = new ArrayList<Edge>();
        }
3       graph[0].add(new Edge(s:0, d:2));

4       graph[1].add(new Edge(s:1, d:2));
        graph[1].add(new Edge(s:1, d:3));
5
        graph[2].add(new Edge(s:2, d:0));
6       graph[2].add(new Edge(s:2, d:0));
        graph[2].add(new Edge(s:2, d:0));

        graph[3].add(new Edge(s:3, d:1));
        graph[3].add(new Edge(s:3, d:2));
    }

    Public Static void main(String args[]) {
    int v = 4;
    ArrayList<Edge> graph[] = new ArrayList[v];
}}
```

Now, there are 2 major reasons to use the adjacency list to store the graph info. They are as follows —

(i) No extra space is required. To store the graph, we are not utilizing any extra memory space.

(ii) Time is optimized to find neighbours, and time complexity comes out to be $O(x)$, where x = no. of neighbours.

For e.g., in this example, (2) has 3 neighbours. So, time complexity should be $O(3)$, and so on.

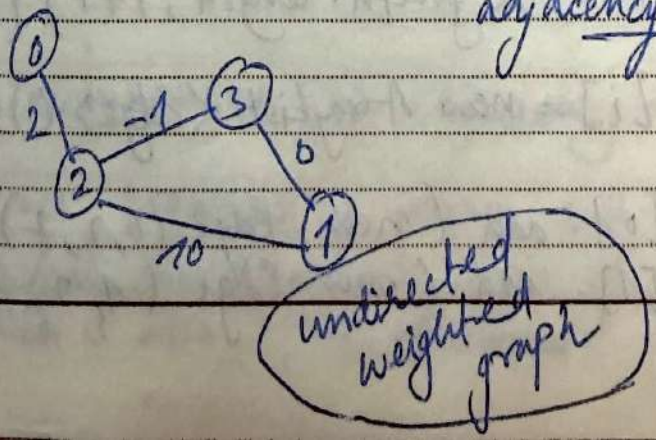
```
for(int i=0 to graph[2].size())
    Edge e = graph[2].get(i);
    Print(e.dst); // neighbours.
```

Edge e → src, dst.

```
// print 2' neighbours
createGraph(graph); // call createGraph func. initially
for(int i=0; i<graph[2].size(); i++) {
    Edge e = graph[2].get(i);
    System.out.print(e.dest + " ");
}
```

Example 2

Create a graph using adjacency list



11 October
42nd Wk • 284-081
Tuesday

2022 OCTOBER

M	T	W	T	F	S	S	M	T	W	T	F	S	S
					1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31						

So, for this example, we have 3 properties of edge, ie source, destination & weight.

ArrayList <Edge> Graph[]
Edge = (src, dst, weight)

0 → {0, 2, 2}
1 → {1, 2, 10}, {1, 3, 0}
2 → {2, 0, 2}, {2, 1, 10}, {2, 3, -1}
3 → {3, 1, 0}, {3, 2, -1}

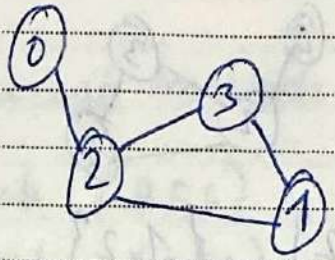
→ static class Edge {
    int src;
    int dest;
    int wt;

    Public Edge (int s, int d, int w) {
        this.src = s;
        this.dest = d;
        this.wt = w;
    }
}

Public static void createGraph (ArrayList <Edge> graph[])
{
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList <Edge> ();
    }

    graph[0].add (new Edge (0, 2, 2));
    graph[1].add (new Edge (1, 2, 10));
    :
}

Adjacency Matrix



undirected unweighted graph

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

(0,1)

(i,j)

0 → i→j (X)

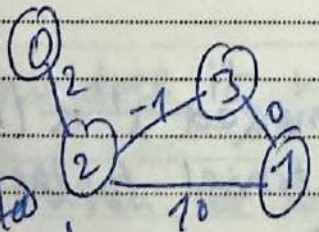
1 → i→j (✓)

'0' meaning no edge exists.

'1' meaning edge exists ✓

The dimension of the matrix depends upon the no. of vertices. Here, V=4, that's why we made a 4×4 matrix.

If the graph is a weighted graph, then we can directly store the weight (in place of 1).



undirected weighted graph

	0	1	2	3
0	0	0	2	0
1	0	0	10	0
2	2	10	0	-1
3	0	0	-1	0

Disadvantages of adjacency matrix

(i) Space required is more and unnecessary. $O(V^2)$

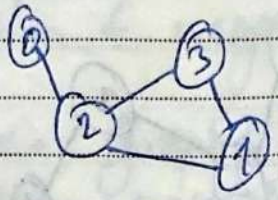
(ii) To find the neighbour, time of $O(V)$ is required which is also not optimised.

13 Thursday October
42nd Wk • 286-079

2022 OCTOBER

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Edge list-



Edges = { {0,2}, {1,2}, {1,3}, {2,3} }

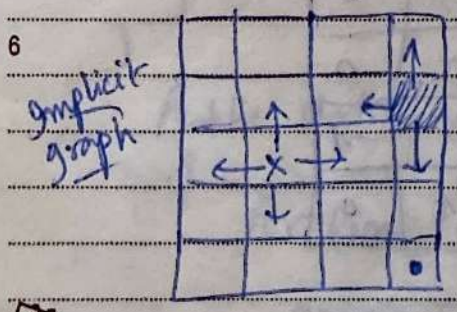
we can store the edge related information in the form of arraylist or linkedlist.

usually, in scenario where we need to sort the edges based on weight or in case of Minimum Spanning Tree (MST), we prefer to use edge list.

Implicit graph

Implicit graph is nothing but the 2D array itself. If, in our problem, a 2D graph is already given and we need to traverse from a cell to another, we can use this approach.

Such an algorithm, where the implicit graph can be used is flood fill algorithm.



Suppose the cell marked 'x' is (i,j) we are told to travel to the last cell. Therefore,

(i-1,j)
(i,j-1) ← (i,j) → (i,j+1)
(i+1,j).

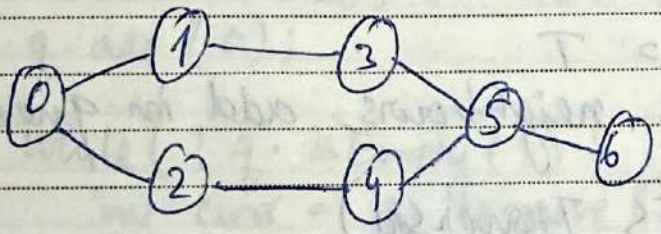
So, every cell shall have 4 neighbours, except the edge cell having 3 neighbours.

** Graph Traversals **

There are 2 common traversal methods —

(I) Breadth First Search (BFS)
(II) Depth First Search (DFS).

Breadth First Search (BFS)



unlike tree, graphs doesn't have a starting point such as root. By convention, we can choose a starting point (e.g → 0).

Now, the rule is "Go to immediate neighbours first."

0 → 1 → 2 → 3 → 4 → 5 → 6.

How does it work?

(I) when we are 0, we have two choices → 1 & 2. So, we go to any one of them, let's say '1'. Now, before going to anywhere else, all immediate neighbours of 1 has to be covered.

So, 0 → 1 → 2. Then immediate neighbour of 1, so we go to 3. So, 0 → 1 → 2 → 3.

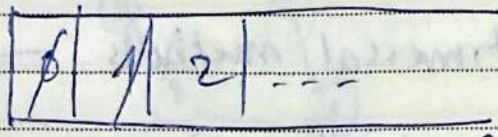
We always need to check if the immediate neighbours of previous nodes are already visited or not.

15 October
42nd Wk • 288-077
Saturday

2022 OCTOBER

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

The data has to be stored in a queue.

Queue  FIFO.

Front Rear

The visiting of the node goes in the following –

if (vis[curr] == F),

1. Print (curr)
2. vis[curr] = T
3. curr. node neighbours, add in queue.

Pseudo Code (BFS Traversal)

```
while(!q.empty()) {
int curr = q.remove()
    if(vis[curr]==F){
        1. print(curr)
        2. vis[curr] = T
        3. curr. node neighbours add in Q
    }
for(int i=0 to graph[curr].size())
    Edge e = graph[curr].get(i);
    q.add(e.dest); // add neighbour
```

create Q
create vis()
Q.add(0)
initialization

Time complexity (BFS): O(V+E).

If $V >> E$, then $O(V)$ or else if too many edges, then $O(E)$.

But in general time complexity is $O(V+E)$.

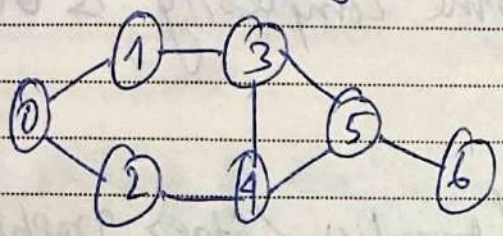
Java code :-

```
Public Static void bfs (ArrayList <Edge> graph[], int v)
{
    Queue <Integer> q = new LinkedList <> ();
    boolean vis[] = new boolean [v];
    q.add (0);

    while ( ! q.isEmpty ()) {
        int curr = q.remove ();
        if (vis [curr] == false) {
            System.out.print (curr + " ");
            vis [curr] = true;

            for (int i=0; i< graph [curr].size(); i++) {
                Edge e = graph [curr].get(i);
                q.add (e.dest);
            }
        }
    }

Public Static void main (String args[]) {
    int v = 7;

    ArrayList <Edge> graph[] = new ArrayList [v];
    createGraph (graph);
    bfs (graph, v);
}
```

** Note :- createGraph & Edge written in 295. **

DFS (Depth First Search)

"Keep going to the 1st neighbour".



the main difference between BFS & DFS is that —

In BFS, the priority is to traverse all immediate neighbours. so, if we start from (0), we go to (1). At this point, priority is not to travel to the neighbour of (1), but to complete the immediate neighbour of (0). So, next we go to (2). So, $0 \rightarrow 1 \rightarrow 2$.

In DFS, the priority is to keep going to 1st neighbour of the next node. so, we start from (0). Then we go to (1). At this point, we check the next neighbour of (1) and travel to (3). so, $0 \rightarrow 1 \rightarrow 3$.

like this, $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$.

- Start at 0. Go to next node, ie 1.
- Now check neighbours of 1. '0' is already printed, so go to 3. $0 \rightarrow 1 \rightarrow 3$.
- Then check neighbours of 3. '1' is printed. Go to 4. $0 \rightarrow 1 \rightarrow 3 \rightarrow 4$.
- check neighbour of 4, i.e 2, $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2$

Now, neighbour of 2 is 0 & 4, both are printed. come back to 4. Go to 5, $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

- check neighbour of 5. 3, 4 already printed, so, 6.

$\therefore 0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$

Note :- 1 major difference between graph traversal and tree traversal is that -

✓ In tree traversal, we don't have cyclic traversal, because tree is a hierarchical data structure. we can't just jump from the leafs to the root.

✓ whereas, in graph traversal, we have several cyclic traversals and that's why we use the vis[] (visited array), to check whether we have travelled to a specific node or not.

Pseudo code,

```
void dfs ( graph curr, vis[] ) {
    ① print (curr)
    ② vis[curr] = T
    ③ for (int i = 0 to graph[curr].size())
        Edge e = graph [curr]. get(i);
        dfs ([graph, e. dst, vis) ;
    }
```

Time complexity of DFS = time complexity of BFS

= O(V+E).

Java code :-

```
8  Public static void dfs (ArrayList<Edge> graph[], int curr,
                           boolean vis[]) {
9      System.out.print (curr + " ");
       vis[curr] = true;
10
       for (int i=0; i< graph[curr].size(); i++) {
11         Edge e = graph[curr].get(i);
           if (vis[e.dest] == false) {
12             dfs (graph, e.dest, vis);
           }
       }

1  Public static void main (String args[]) {
       int v = 7;
       ArrayList<Edge> graph[] = new ArrayList[v];
2      createGraph (graph);

3      boolean vis[] = new boolean[v];
       dfs (graph, 0, vis);
4      System.out.println();
   }
5  }
```

6 note :- Creation of Edge class, createGraph() is already done. (Pg - 295)

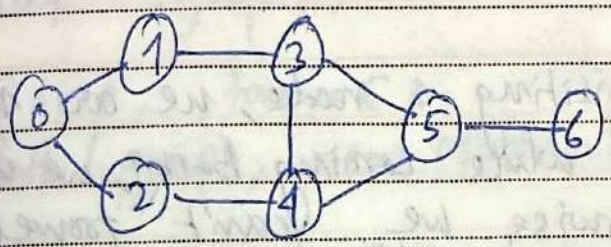
// If we have disconnected graphs with several components,

```
boolean vis[] = new boolean[v];
for (int i=0; i<v; i++) {
    if (vis[i] == false) {
        dfs (graph, 0, vis);
    }
}
```

Q) All Paths from Source to Target

For given src & tar, print all the paths that exist from src to tar.

E.g.,



src = 0
tar = 5

ans:- 0 → 1 → 3 → 4 → 5
0 → 1 → 3 → 5
0 → 2 → 4 → 3 → 5
0 → 2 → 4 → 5

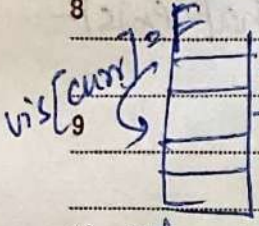
Pseudo code

```
dfs( graph, vis[], curr, String Path, tar)
    if(curr = tar)
        Print (path);
        return;

    for (int i=0 to graph[curr].size()) {
        Edge e = graph[curr].get(i);
        if( vis[e.dest] == F) {
            vis[curr] = T;
            dfs( graph, vis, e.dest, Path+e.dest, tar);
            vis[curr] = F;
        }
```

21 October
43rd Wk • 294-071
Friday

2022 OCTOBER

So, here we are basically using the DFS algorithm, but in a modified way.



Whenever we are visiting a node, we are making it as True, whereas while coming back, we are making it as False, otherwise we can't travel it again. Here, same node can be travelled many times to find all paths.

Also, we need to use the vis[] array, otherwise it may happen that the code may get stuck in an infinite loop while travelling back and forth between 2 nodes.

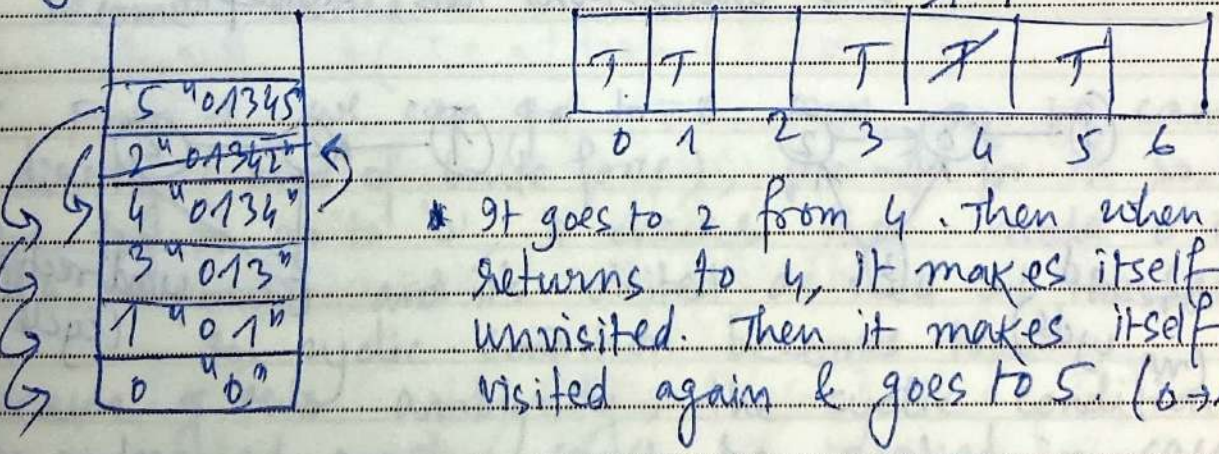
Java code :-

```
Public static void printAllPath (ArrayList<Edge> graph[],
    boolean vis[], int curr, String path, int tar) {

    if (curr == tar) {
        System.out.println(path);
        return;
    }

    for (int i=0; i<graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        if (!vis[e.dest]) {
            vis[curr] = true;
            printAllPath (graph, vis, e.dest, path+e.dest, tar);
            vis[curr] = false;
        }
    }
}
```

```
public static void main (String args[]) {
    int v = 7;

    ArrayList <Edge> graph[] = new ArrayList [v];
    createGraph (graph);

    int src = 0, tar = 5;
    printAllPath (graph, new boolean[v], src, "0", tar);
}
```

note:- Though search time complexity in DFS is O(V+E), which is very optimized, the time complexity in printAllPath() is O(V^V), which is not very optimized. But we need to travel all nodes several times.

Dry run:-



* It goes to 2 from 4. Then when it returns to 4, it makes itself unvisited. Then it makes itself visited again & goes to 5. (0→1→3→4→5).

* Also, when the path is traversed we print it and return. So, it from 5, it returns to 4 and the stack with '5' is deleted from the stack.

Then, '4' returns to '3' and it's also deleted. Now when we are in '3', firstly it makes itself unvisited. Then it checks it's neighbours. Both '1' & '4' are already visited, so it directly calls for '5'.

So, 0→1→3→5. Same for other paths.

** Cycle Detection **

Directed Graphs (DFS)

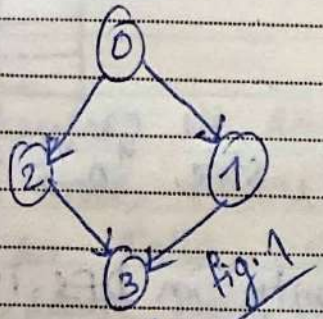


Fig. 1

no cycle

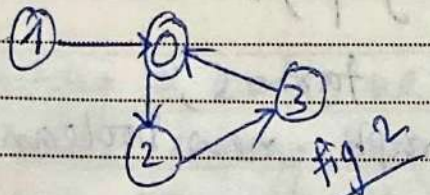
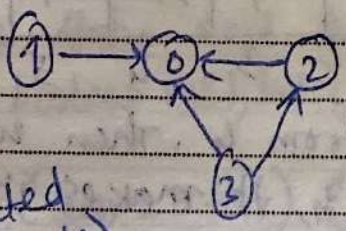


Fig. 2

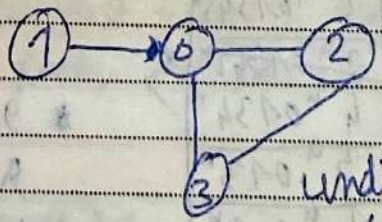
cycle

Now, the thing that we need to understand is that the approach in the undirected graph shall not work in case of directed graphs.

Let's understand this concept.



directed (no cycle)



undirected (cycle)

undirected graph,

✓ Let's start with 0, Parent (P) = -1. It will search it's next node, and move to 1. For 1, P=0. From 1, we shall move back to 0. (0 & 1 both visited)

✓ Now, we move to 2. For 2, P=0. Next we make 2 as visited and move to 3. For 3, P=2. Now, 3 has 2 neighbours. When it checks 2, it's the parent and visited. Next, it checks 0, which is not it's parent but visited. Then we return the cycle condition as true.

310

So, basically when the code is trying to visit a node (here, the node 0) from another node (here, node 3). DFS checks the condition of cycle :-

- ✓ If it's Parent or not (shall not be parent)
- ✓ if it's already visited (shall be visited)

Now, if we apply the same logic to ~~cyclic graph~~ directed graph,

Directed graph

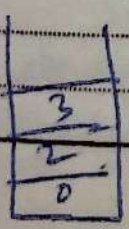
✓ we start at 0. So, Parent (P) = -1. Now from node 0, we cannot move anywhere.

✓ So, we move to 1 and start again, directly travelling from 0. At this point, '0' has become visited.

✓ From 1, we can go to 0. Now, as we came directly (kind of brute force), P = -1 for 1. So, when we try to go to '0', it checks that node '0' is not it's parent and it's visited as well at the same time. So, cyclic condition becomes tone.

Because of this ambiguity, the cyclic condition of the undirected graph can't be applied in case of directed graph.

That's why we are going to use a modified DFS to determined the cyclic condition in case of the directed graphs.



For fig. 2. we start with 0. next go to 2. next from 2, go to 3. next when we try to visit 0, it checks the stack and returns 'tone' for cyclic condition.

Basically, we are using the concept of recursion stack. Also, we shall not use any 'parent' concept. So, we keep building the stack with every visit to a node.

Pseudo code

```
dfs ( graph, vis[], curr, rec[])
        vis[curr] = T
        rec[curr] = T

for(int i=0 to graph[curr].size()) {
        edge e = graph[curr].get(i);
    if(rec[e.dest] == T){   // cyclic condition
        return T; }
    else if(!vis[e.dest]) {
        dfs(graph, vis, e.dest, rec[]);}
    }
    rec[curr] = F;
}
```

Java code :-

```
import java.util.*;
public class Classroom {
    static class Edge {
        int src;
        int dest;

        Public Edge (int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }
    Public static void createGraph (ArrayList <Edge> graph[]){

        for (int i=0; i<graph.length; i++) {
            graph[i] = new ArrayList <Edge> ();
        }
        graph[0].add (new Edge (s:0, d:2));
        graph [1].add (new Edge (s:1, d:0));
        graph [2].add (new Edge (s:2, d:3));
        graph [3].add (new Edge (s:3, d:0));
    }

    Public static boolean isCycleDirected (ArrayList <Edge> graph[],
            boolean vis[], int curr, boolean rec[]) {

        vis[curr] = true;
        rec[curr] = true;
```

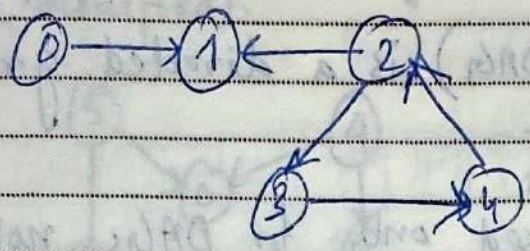
27 October
Thursday
44th Wk • 300-065

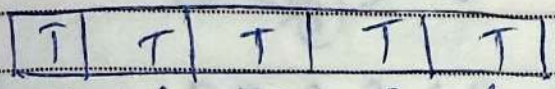
2022 OCTOBER

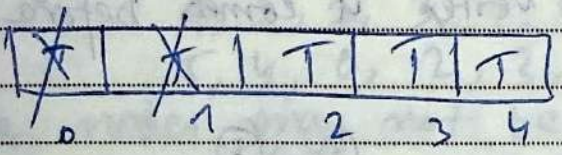
M	T	W	T	F	S	S	M	T	W	T	F	S	S
					1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31						

```
for (int i=0; i<graph[curr].size(); i++) {
    Edge e = graph[curr].get(i);
    if (rec[e.dest] == true) { // cycle condition.
        return true;
    } else if (!vis[e.dest]) {
        if (isCycleDirected(graph, vis, e.dest, rec))
            return true; }
    }
    rec[curr] = false;
    return false;
}
Public static void main (String args[]) {
    int v=4;
    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);
    System.out.println(isCycleDirected(graph,
        new boolean[v], curr: 0, new boolean[v]));
    }
}
// if the graph is broken into components :-
// inside main() function
boolean vis[] = new boolean[v];
boolean rec[] = new boolean[v];
for (int i=0; i<v; i++) {
    if (!vis[i]) {
        boolean isCycle = isCycleDirected(graph, vis, 0, rec);
        if (isCycle) {
            System.out.println(isCycle);
            break; }}}
```

Let's dry run the code with another example :-







1. Start with 0. Make vis = T, rec = T. Move to 1.

2. At 1, Make vis = T, rec = T. Now from 1, we can't move anywhere. So, we go back to 0. while going back, we remove it from rec. stack.

3. We come back to 0. Now, from 0, we can't go anywhere else. Everything is visited. So, we move to an unvisited node. while going to an unvisited node, we remove it from the rec. stack.

4. Now, we are at 2. As we reach 2, vis = T, rec. Stack = T. From 2, we go to 3. As we reach 3, vis = T, rec. stack = T.

5. From 3, we go to 4. As, we reach 4, we make vis = T, rec. Stack = T. Now, from 4, we see it's neighbours. Now, when we try to move to 2, we check that vis = T as well as rec. stack = T. So, it returns to 3, cycle = T. Then 3 returns to 2 as true and then to 1 as true. and finally 1 returns to main function as true.

29 October 2022
Saturday
44th Wk • 302-063

OCTOBER
M T W T F S S M T W T F S S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

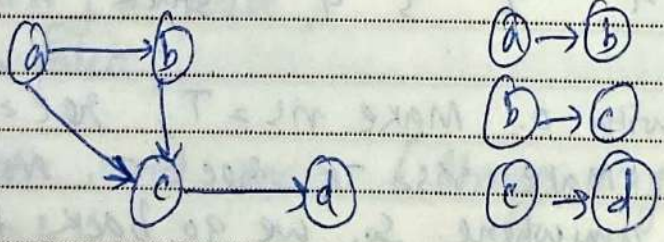
Topological Sorting

Directed Acyclic Graph (DAG) is a directed graph with no cycles.

Topological Sorting is used only for DAGs, not for Non-DAGs.

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

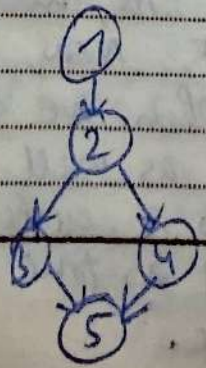
E.g.



So, we can say abcd is my topological order for this given graph. 'bcad' or 'dcab' or otherwise can not be the topological order.

Another example

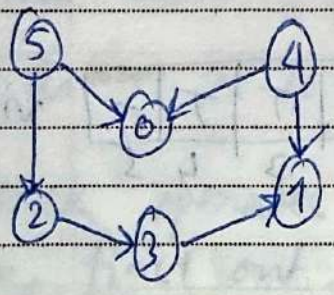
Action 1 → buy laptop
Action 2 → install OS
Action 3 → install code editor
Action 4 → install Java
Action 5 → write code



1 2 3 4 5 or 1 2 4 3 5.

It actually shows the dependency, i.e which action is dependent on which previous action.

Now, to implement topological sorting, we shall the method of DFS (modified).

E.g., 

In this example, there can be multiple answers possible.

Ans:- 5, 4, 2, 3, 1, 0
4, 5, 2, 3, 1, 0
5, 4, 0, 2, 3, 1, etc.

The nodes may not be in chronological order but we need to keep in mind that the dependant node should come later in the order than the node on which it is dependant. For e.g., (3) should always come after (2), (1) should always come after (3) & (4), etc.

Pseudo code ≡→

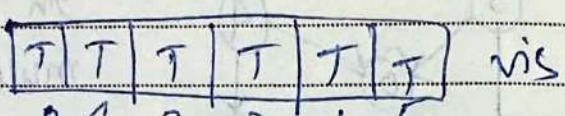
```
for (int i = 0 to V)
{ if (vis[i] == F)
    dfs(..);
```

```
dfs (graph, vis, curr, Stack)
vis[curr] = T
for (int i = 0 to graph[curr].size()) {
    Edge e = graph[curr].get(i)
    if (! vis[e.dest])
        dfs (graph, vis, e.dest, Stack)
}
Stack.push(curr); // only change from normal dfs code.
}
```

(further part if it has neighbours)

Stack.pop();

Dry run:-



Steps:-

1. visit 0. Now, make vis[0] = T. As '0' has no neighbours, so, the further part of code shall not run. Now, while going back, we push '0' to the stack.

2. Same goes for 1. we go to 1, make vis[1] = T. '1' doesn't have neighbours, so we go back and push it to the stack.

3. Next unvisited node is 2. So, vis[2] = T. Now '2' has neighbours, i.e '3'. So, we visit 3. make vis[3] = T. Now, we try to visit of 3, which is 1, but it's already visited. So, we will not visit and make any call. '3' doesn't have other neighbours. We go out and push '3' to stack.

Now, we are back to '2'. 2 doesn't have any other neighbour. so, we go back and push '2' to the stack.

4. Next unvisited is 4. we visit '4' and check neighbours. Both neighbours '2' & '1' are visited, so no call made. Push '4' to stack.

5. Next unvisited is 5. other neighbours are visited. Push '5' to stack.

01 Tue 02 Wed 03 Thu 04 Fri

6. Now, once all the elements are visited, we move out of the outer loop. So, the last thing we need to do is to pop out the elements.

05 Sat 06 Sun 07 Mon 08 Tue

Stack operates in LIFO fashion. So, last in, first out.

09 Wed 10 Thu 11 Fri 12 Sat

First 5, then 4, then 2, and so on

$5 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$.

13 Sun 14 Mon 15 Tue 16 Wed

This becomes my topological sorting order for the given graph.

17 Thu 18 Fri 19 Sat 20 Sun

The logical that it's follows is that → It will add the neighbouring (dependant) elements (nodes) first in the stack, and then it will add.

21 Mon 22 Tue 23 Wed 24 Thu

So, when push out the elements, it becomes automatically a topological sorted list.

25 Fri 26 Sat 27 Sun 28 Mon

E.g → 0, 1 added, then only 4 gets added.
Similarly, 3 added, then only 2 gets added.

29 Tue 30 Wed

Note:- Time complexity is $O(V+E)$.

NOV

Java code :-

```
import java.util.*;
Public class Classroom {
    static class Edge {
        int src;
        int dest;
    Public Edge (int s, int d) {
        this.src = s;
        this.dest = d;
    }
}
Public Static void CreateGraph(ArrayList<Edge> graph[])
{
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList<Edge>();
    }
    graph[2].add(new Edge(s:2, d:3));
    graph[3].add(new Edge(s:3, d:1));
    graph[4].add(new Edge(s:4, d:0));
    graph[4].add(new Edge(s:4, d:1));
    graph[5].add(new Edge(s:5, d:0));
    graph[5].add(new Edge(s:5, d:2));
}
Public static void topSort(ArrayList<Edge> graph[], int curr,
    boolean vis[], Stack<Integer> stack) {
```

```
        vis[curr] = true;
8       for (int i=0; i<graph[curr].size(); i++) {
            Edge e = graph[curr].get(i);
9           if (!vis[e.dest]) {
10              topSort(graph, e.dest, vis, stack);
            }
11      }
            stack.push(curr); //only change
12  }

    Public static void topSortUtil(ArrayList<Edge> graph[], int v)
1   {
        boolean vis[] = new boolean[v];
2       Stack<Integer> stack = new Stack<>();

3       for (int i=0; i<v; i++) {
            topSort(graph, i, vis, stack);
4       }
        }
5       while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
6       }
    }
    Public static void main(String args[]) {
        int v=6;
        ArrayList<Edge> graph[] = new ArrayList[v];
        createGraph(graph);

        topSortUtil(graph, v);
    }
}
```

03 November
45th Wk • 307-058
Thursday

2022 NOVEMBER

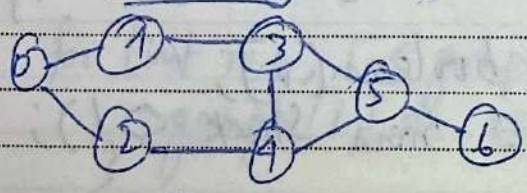
M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

** cycles in Graphs **

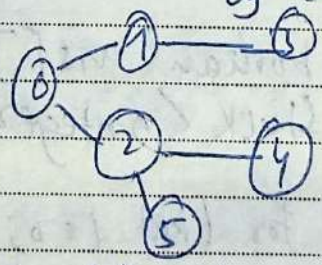
8 undirected graphs → DFS
→ BFS
9 → DSU (Disjoint Set Union)

10 Directed graph → DFS
→ BFS
11 → Topological sorting (Kahn's algo)
12 → graph coloring.

(undirected graph)

1


cycle = true

cycle = false


2 let's look at an example to determine the
4 cyclic detection —

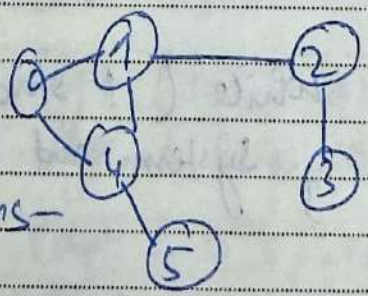
5


fig 1.

For a given neighbour,
6 there are basically 3 conditions—

1) vis [n] = T
Parent = F ⇒ cyclic condition.

2) vis [n] = T
Parent = T ⇒ no need to take action

3) vis [n] = F ⇒ visit it.

The cyclic condition becomes true when the node that we are going to visit is already visited and it's not the parent node also.

For E.g, (fig 1.)

→ start with 0. Parent = -1. vis[0] = true. go to 1.
At 1, vis[1] = 0, Parent = 0. go to 2.
→ At 2, vis[2] = T, Parent = 1. go to 3. vis[3], Parent = 2. No new neighbours of 3. come back to 2.
2 has no other neighbour except 3, come back to 1.
1 has neighbours 2, & 0, both are already visited. 1 also has '4'.

→ now, go to next unvisited node, i.e 4. Here vis[4] = T.
Parent = 1. Now 4 has 3 neighbours - 0, 1, 5.

For neighbour 5, → we will visit as vis[5] = F.
For neighbour 1, → vis[1] = T, P = T, so no action.
For neighbour 0, → vis[0] = T, P = F. Cyclic exists.

So, 4 shall return to 1 as cycle = T. 1 shall return to 0 for cycle condition = T and '0' shall return to main as cycle condition = T.

Pseudo code

```
dfs(graph, vis[], curr, par)
    vis[curr] = T.
    for (int i=0 to graph[curr].size())
        Edge e = graph[curr].get(i)
(1) if (vis[e.dest] = T && par != e.dest) return true;
(3) if (!vis[e.dst]) {
        if (dfs(graph, vis, e.dest, curr) == T) { return true; }}
```

Java code :-

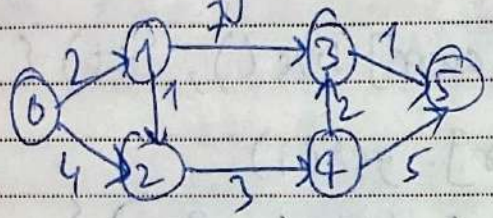
```
import java.util.*;

Public class Classroom {
    static class Edge {
        int src;
        int dest;
        Public Edge (int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }
    Public static void createGraph (ArrayList<Edge> graph[])
    {
        for (int i=0; i< graph.length; i++) {
            graph[i] = new ArrayList <Edge> ();
        }
        graph[0].add (new Edge (s:0, d:1));

        graph [1].add (new Edge (s:1, d:0));
        graph [1].add (new Edge (s:1, d:2));
        graph [1].add (new Edge (s:1, d:4));

        graph [2].add (new Edge (s:2, d:1));
        graph [2].add (new Edge (s:2, d:3));

        graph [3].add (new Edge (s:3, d:2));

        graph [4].add (new edge (s:4, d:0));
        graph [4].add (new edge (s:4, d:1));
        graph [4].add (new edge (s:4, d:5));

        graph [5].add (new Edge (s:5, d:4)); }
```

```
Public static boolean isCycleUndirected (ArrayList<Edge> graph[],
        boolean vis[], int curr, int par) {

    vis[curr] = true;
    for (int i=0; i< graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        if (vis[e.dest] && e.dest != par) {
            return true;
        }
        else if (!vis[e.dest]) {
            if (isCycleUndirected(graph, vis, e.dest, curr)) {
                return true;
            }
        }
    }
    return false;
}

Public static void main (String args[]) {
    int V = 6;
    ArrayList<Edge> graph[] = new ArrayList[V];
    createGraph(graph);

    System.out.println(isCycleUndirected(graph, new boolean[V],
                        curr:0, -1));
}
}
```

Note :- Time complexity of this cyclic detection = $O(V+E)$.
It's same as the time complexity of DFS.

** Shortest path algorithms **

* Dijkstra's Algorithm * (Time complexity = O(E + E log V))



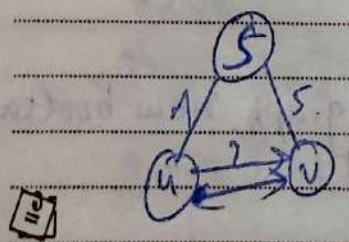
This algorithm helps us to find the shortest distance from the source to all vertices.

dest	0	2	3	8	6	9
	0	1	2	3	4	5

0 to 0 → starting point is equal to dest.
0 to 1 → only 1 feasible way (direct) 0 → 1.
0 to 2 → 0 → 1 → 2
0 to 3 → 0 → 1 → 2 → 4 → 3
0 to 4 → 0 → 1 → 2 → 4
0 to 5 → 0 → 1 → 2 → 4 → 3 → 5

In this manner, we find the shortest distance (from src) to a particular node.

Now, before diving deep into the shortest path algorithm, we need to know a concept of relaxation.



dist [v] (src to v) = ∞ 3

dist [u] (src to u) = 1

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

So basically, if we find a way where it takes the shortest time, even shorter than the pre-determined direct path, then we update the dist to the new value. This process of updating is called Relaxation.

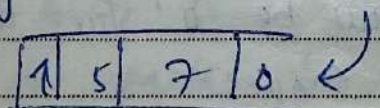
So, dist[u] + wt = 1 + 2 = 3 < dist[v] = 5

So, dist[v] = 3 (updated)

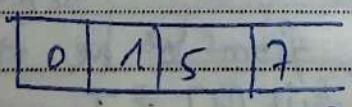
→ Dijkstra's algorithm is called greedy algorithm.

→ Usually, it implements BFS algorithm. We shall also try to implement it with a bit of modification.

→ BFS implements Queue. Here, we shall use Priority Queue, which internally sorts the value based on priority.

Eg, 

Queue

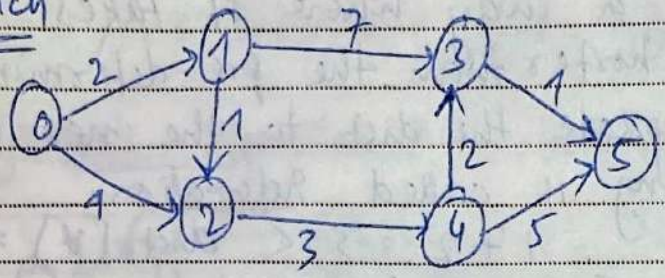


Priority Queue

Based on priority, it sorts out the elements internally. After 1, 5, 7, when '0' was added, it was added at the front because it has the lowest value, so highest priority.

Let us know see how Dijkstra's algorithm can be implement. So, we shall discuss the approach, then the pseudo code and finally the Java code.

Approach



nodes
→ visit → false
→ dist → shortest

dist: | 0 | ∞ 2 | ∞ 4 3 | ∞ ∞ 8 | ∞ 6 | ∞ 11 9 |
0 1 2 3 4 5

visit:- | F T | ∞ T | ∞ F T | ∞ F T | ∞ F T | ∞ F T |
0 1 2 3 4 5

1. At the initial stage, the visit[] shall be false (F) for all the cells.
The dist[] shall be '0' for the src and ∞ for the rest of the nodes.

2. We shall check 2 things for the nodes. → i) visit = false, ii) dist = shortest. Now, from '0' we visit the neighbours. We go to 1. We make visit[1] = T and update the dist from '∞' to '2'. Similarly for 2 also, update dis[2] = 4, which is the direct distance from src. So, all neighbour dist. of '0' are updated.

3. From '0' we check (visit = F, dist = shortest) That's why next we visit 1 (because dist 2 < 4) instead of 2.

3. From '1', we make vis[1] = T. Now, we try to update the distance of it's neighbours. From '1' when we check for 2, we perform the operation of relaxation.

So node 1 = u, node 2 = v.

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

Here, 2+1 < 4, so, dist[v] = 3 is update.
So, we update dist[2] = 3 in the dist[]. The other neighbour is 3. From '1' to 3, the path is 0→1→3.
So, we update the dist[3] to 9.

3. Now, we again check node where vis[] = false, distance = shortest. currently, the dist[] looks like -

dist.	0	2	3	9	∞	∞
	0	1	2	3	4	5

So, we go to 2. Now, we make vis[2] = T. And, now, try to update the dist. of it's neighbours. It has only one neighbour - '4'. So, dist[4] = 6, is updated. Now,

dist.	0	2	3	9	6	∞
	0	1	2	3	4	5

So, currently, we again check nodes → visit = false
→ dist = shortest.
So, we go to 4. we make visit[4] = T. And, we update the dist. of neighbours. '4' has two neighbours.
dist[3] is updated to 8. (via 4→3)
dist[5] is updated to 11. (via 4→5).

4. Now, based on vis=F & shortest distance, we visit 3. we make vis[3] = T. From 3, we only have one neighbour, i.e 5. So, we update the distance, dist[5] = 9. Now, we go to 5, since vis[5] = F and update it, vis[5] = T. '5' has no neighbours. So, finally dist. is →

dist. →	0	2	3	8	6	9
	0	1	2	3	4	5

We shall make use of PQ (priority Queue) for this problem. So, basically we need some parameters to decide which shall come first in the priority Queue.

Pseudo code — int, int
Pair (node, dist)

```
class Pair
    PQ.add([0,0])
    dist[]; // initialization → [0|∞|∞|∞|∞|∞]  (src)
    vis[];
while(!pq.isEmpty()){
    Pair curr = pq.remove(); → if(!vis[curr.node]){
    vis[curr.node] = T
    for(int i=0 to ___) // neighbours
        Edge e = graph[curr.n].get(i);
        u → e.src
        v → e.dest
        if(dist[u] + e.wt < dist[v])
            dist[v] = dist[u] + e.wt;
            pq.add(v, dist[v]);
    }}
```

Java code:-

```
import java.util.*;
public class Classroom {
    static class Edge {
        int src;        // edge source
        int dest;       // edge destination
        int wt;         // edge weight
```

```
Public Edge (int s, int d, int w) {
    this.src = s;
    this.dest = d;
    this.wt = w;
  }
}

Public Static void createGraph (ArrayList<Edge> graph[])
{
  for (int i=0; i< graph.length; i++) {
    graph[i] = new ArrayList<Edge>();
  }
  graph[0].add(new Edge(s:0, d:1, w:2));
  graph[0].add(new Edge(s:0, d:2, w:4));

  graph[1].add(new Edge(s:1, d:3, w:7));
  graph[1].add(new Edge(s:1, d:2, w:1));

  graph[2].add(new Edge(s:2, d:4, w:3));

  graph[3].add(new Edge(s:3, d:5, w:1));

  graph[4].add(new Edge(s:4, d:3, w:2));
  graph[4].add(new Edge(s:4, d:5, w:5));

}

Public Static void dijkstra(ArrayList<Edge> graph[],
                            int src)
{
```

```
PriorityQueue<Pair> pq = new PriorityQueue<>();
int dist[] = new int[v];
for(int i=0; i<v; i++){
    if(i != src){
        dist[i] = Integer.MAX_VALUE;
    }
}
boolean vis[] = new boolean[v];
pq.add(new Pair(n:0, d:0));

while(!pq.isEmpty()){

    Pair curr = pq.remove(); // shortest
    if(!vis[curr.node]){

        vis[curr.node] = true;

    for(int i=0; i<graph[curr.node].size(); i++){
        Edge e = graph[curr.node].get(i);
        int u = e.src;
        int v = e.dest;

        if(dist[u] + e.wt < dist[v]){ //Relaxation

            dist[v] = dist[u] + e.wt;
            pq.add(new Pair(v, dist[v]));
        }
    }
}
```

```
for (int i=0; i<V; i++) {
    System.out.print(dist[i] + " ");
}
System.out.println();
}

Public static class Pair implements Comparable<Pair>{
    int node;
    int dist;

    Public Pair(int n, int d) {
        this.node = n;
        this.dist = d;
    }
    @override
    Public int compareTo(Pair p2) {
        return this.dist - p2.dist; //ascending
    }
}

Public static void main(String args[]) {
    int V = 6;

    ArrayList<Edge> graph[] = new ArrayList[V];
    createGraph(graph);

    dijkstra(graph, src:0, V);
}
}
```

Note:- Time Complexity: $O(E + E \log V)$

$O(E)$ → to traverse each edge

$O(E \log V)$ → for priority queue to sort the pairs.

* Bellman Ford Algorithm * (Time Complexity = O(V·E))

Shortest distance from the source to all vertices. It is similar to Dijkstra's algorithm but it also works in cases whereas, Dijkstra's algorithm doesn't work.

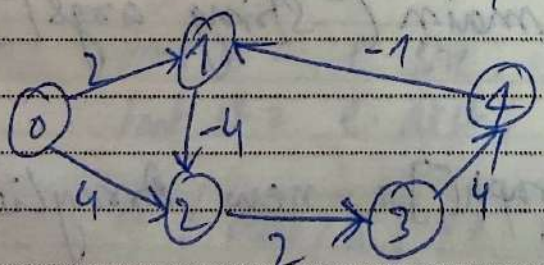
For. e.g., when edges wt. > 0, Dijkstra's work. However, when edges wt. < 0, then it doesn't work and hence we use Bellman Ford Algorithm. Dijkstra's work on greedy algorithm, whereas Bellman Ford works on DP or dynamic programming (DP).

on another note, Dijkstra's has a lesser time complexity, while Bellman Ford has a greater time complexity. So, in cases where we have a positive edge weight, we would try to use Dijkstra's.

Note:- Shortest distance from source to all vertices.

E.g,

Let's look at an example -



0 → 0 (0)
0 → 1 (2)
0 → 2 (-2)
0 → 3 (0)
0 → 4 (4)

Ans:- 0, 2, -2, 0, 4

Perform this operation (v-1) times, v = vertex.

for all edges (u, v)
if dist[u] + wt[u+v] < dist[v]
dist[v] = dist[u] + wt[u, v].

dist[] = | 0 | ∞ | ∞ | ∞ | ∞ |
0 1 2 3 4

1st = | 0 | 2 | 4-2 | 0 | 4 | (1st = 1st iteration)

2nd = | 0 | 2 | -2 | 0 | 4 |

3rd = | 0 | 2 | -2 | 0 | 4 |

4th = | 0 | 2 | -2 | 0 | 4 |

for (int K = 0 to V-1)
↳ run the loop (V-1) times.

Time complexity :- O(V·E). outer loop runs for V times and inner loop runs for E times. so, the time complexity is more compared to Dijkstra's.

Java code :-

```
import java.util.*;

Public class Classroom {

    Static class Edge {
        int src;
        int dest;
        int wt;

        Public Edge (int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
```

```
Public static void createGraph (ArrayList<Edge> graph[]
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList<Edge>();
    }

    graph[0].add(new Edge(s:0, d:1, w:2));
    graph[0].add(new Edge(s:0, d:2, w:4));

    graph[1].add(new Edge(s:1, d:2, w:-4));

    graph[2].add(new Edge(s:2, d:3, w:2));

    graph[3].add(new Edge(s:3, d:4, w:4));

    graph[4].add(new Edge(s:4, d:1, w:-1));
}

Public static void bellmanFord(ArrayList<Edge> graph[],
                    int src, int V) {
    int dist[] = new int[V];
    for(int i=0; i<V; i++) {
        if(i != src) {
            dist[i] = Integer.MAX_VALUE;
        }
    }

    for(int K=0; K<V-1; K++) {  // O(V), outer loop
        for(int i=0; i<V; i++) {  // O(E), inner loop
```

```
for (int j = 0; j < graph[i].size(); j++) {
    Edge e = graph[i].get[j];
    int u = e.src;
    int v = e.dest;

    if( dist[u] != Integer.MAX_VALUE &&
                dist[u] + e.wt < dist[v]) {

        dist[v] = dist[u] + e.wt;
    }
  }
}
    for (int i = 0; i < dist.length; i++) {
        System.out.print (dist[i] + " ");
    }
    System.out.println( );
}

Public static void main (String args[]) {
    int V = 5;

ArrayList <Edge> graph[] = new ArrayList[V];
CreateGraph (graph);
bellmanFord (graph, src: 0, V);
}
}
```

output :- 0 2 -2 0 4

19 November
47th Wk • 323-042
Saturday

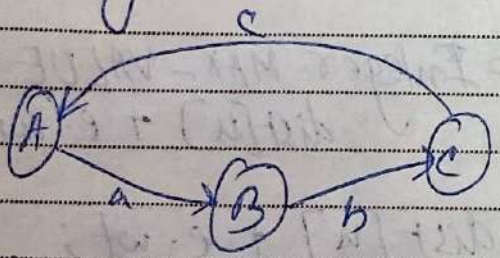
2022 NOVEMBER

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Exception Case :-

Bellman Ford Algorithm doesn't work in negative weight cycles. Because, the weight goes on increasing in the negative direction and then there is no point of finding the shortest distance.

E.g,



If $(a+b+c) < 0 \Rightarrow$ we have got -ve wt. cycle.

To detect whether there's a negative weight cycle or not, we can run the inner loop another time in the Bellman Ford algorithm.

Minimum Spanning Tree (MST)

A minimum Spanning Tree (MST) or minimum weight Spanning tree is a subset of the edges of a connected, edge-weighted, undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight."

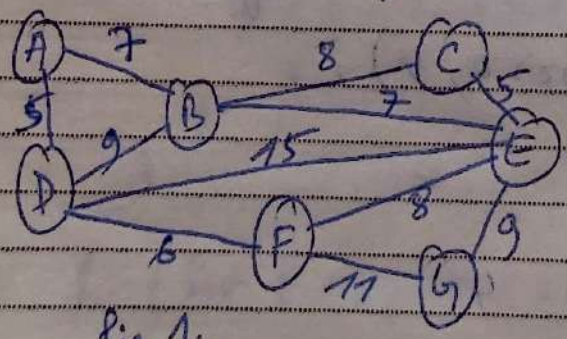
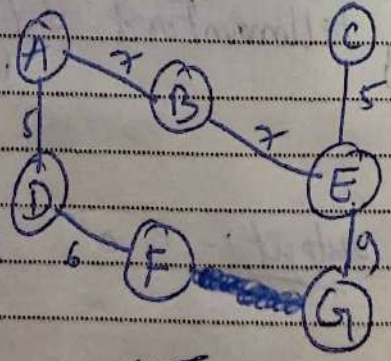


fig 1.

MST



MST of fig 1.

The properties of the MST are as follows —

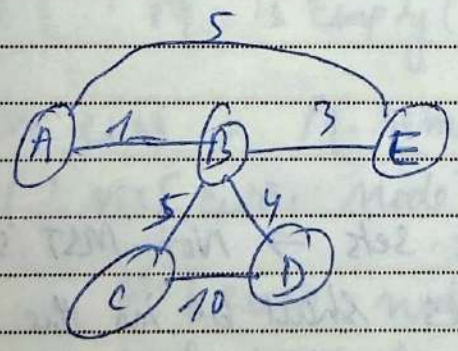
(1) Subgraph of the main graph.

(2) All the vertices are included. All the edges shall not be necessarily included.

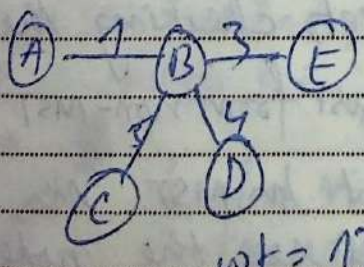
(3) There should be a cycle, unlike typical graph.

(4) one special property is that edge weight (sum of weights of all edges) should be minimum.

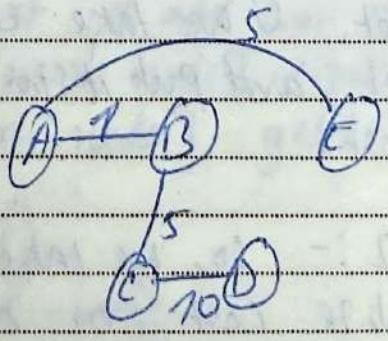
E.g,



Main graph



wt = 13

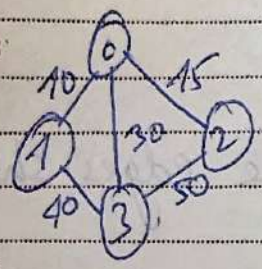


wt = 21

So, left hand tree (min. wt) is the MST of the main graph. There can be multiple Spanning trees of a graph, however there shall be only one MST of a given graph.

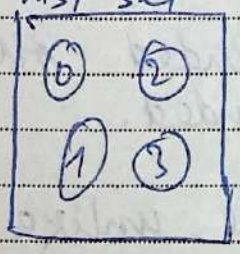
Now, we shall study how we can find out a MST from a given graph, especially with minimum cost.

** Prim's Algorithm (Time complexity = O(E log E)). **



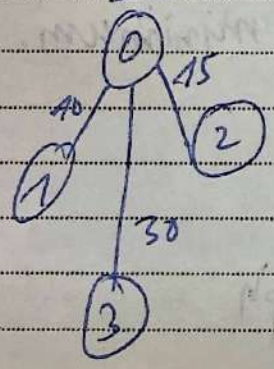
Approach.

MST set



Non-MST set

MST



Cost = 0 + 15 + 10 + 30 = 55.

Step 1:- We shall make 2 sets → Non-MST set & MST set. Initially, all the nodes shall be in the Non-MST set. We take each node from the Non-MST set, and put it in the MST set, checking the minimum cost.

Step 2:- So, we take '0' (min cost from non-MST to MST) and put in MST. Also, we initialize cost = 0. Now, we shall see the node from '0' that shall be of min. cost. From '0' there are 3 nodes - with cost 10, 15, 30. So, we go to 1, and write the cost as well. Next, we take 1 and put it to MST set.

Step 3:- Now from 0 & 1, lowest cost path is 0→2. So, we take 2 and put it to MST set. Same approach we follow for 3 aswell. All while, cost getting included. Everytime we check path from MST set to Non-MST set with minimum cost.

Basically, when we are transferring a node from non-MST set to MST, we are again and again looking for the minimum cost. Hence, we can use Priority Queue (PQ) for the Non-MST set.

Also, if we are taking node to MST set, that means we are already transferring it and visiting it. So, we can use vis[] (visited array) for MST set.

Pseudo code

```
Pq.add( Pair(0,0)) // Pair(node, cost) add to PQ.
boolean vis[] = F

while( ! Pq.isEmpty()) { // until my non-mst set is empty
Pair curr = Pq.remove(); // remove element on min-cost
if( ! vis[curr.node]) {
    vis[curr.node] = T // visit the node
    cost = cost + curr.cost // add cost of edge
    for(int i=0; i< graph[curr.node].size(); i++){
    Edge e = graph[curr.node].get(i);
// now, repeat process for its neighbour
    if (! vis [e.dest])
        Pq.add(e.dest, e.wt);
    }
}
```

Java code :-

```
import java.util.*;

Public class Classroom {
    Static class Edge {
        int src;
        int dest;
        int wt;
        Public Edge (int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    Public Static void createGraph (ArrayList<Edge> graph[])
    {
        for(int i=0; i < graph.length; i++) {
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(s:0, d:1, w:10));
        graph[0].add(new Edge(s:0, d:2, w:15));
        graph[0].add(new Edge(s:0, d:3, w:30));

        graph[1].add(new Edge(s:1, d:0, w:10));
        graph[1].add(new Edge(s:1, d:3, w:40));

        graph[2].add(new Edge(s:2, d:0, w:15));
        graph[2].add(new Edge(s:2, d:3, w:50));

        graph[3].add(new Edge(s:3, d:1, w:40));
        graph[3].add(new Edge(s:3, d:2, w:50));
    }
}
```

```
Public static class Pair implements Comparable <Pair> {
    int node;
    int cost;

    Public Pair (int n, int c) {
        this.node = n;
        this.cost = c;
    }
    @override
    public int compareTo (Pair p2) {
        return this.cost - p2.cost; // ascending
    }
}
Public static void primsAlgo (ArrayList <Edge> graph[], int v) {

PriorityQueue <Pair> pq = new PriorityQueue <>(); //non-mst
boolean vis[] = new boolean [v]; // mst
pq.add (new pair (n:0, c:0));
int mstCost = 0;

    while ( ! pq.isEmpty()) {
        Pair curr = pq.remove();
        if ( ! vis[curr.node]) {
            vis[curr.node] = true;
            mstCost += curr.cost;

        for ( int i=0; i< graph[curr.node].size(); i++){
        Edge e = graph[curr.node].get(i);
        if (! vis[e.dest]) {
                pq.add (new Pair (e.dest, e.wt));
        }}}
        System.out.println ("min. cost of mst" + mstCost);
}
```

```
Public static void main (String args[]) {
    int V = 4;

    ArrayList <Edge> graph [] = new ArrayList[V];
    createGraph(graph);
    primsAlgo (graph, V);
}
}
```

Note :- Time complexity = E log E. It is because in worst case scenario, we add E number of edges inside the priority queue. Then, we sort these E edge inside PQ for min. cost. That's why time complexity is E log E, for sorting 'E' no. of edges.

Dry run

PQ [(0,0) (1,10) (2,15) (3,30) (3,40) (3,50)]

vis

T	T	T	T
0	1	2	3

Pair (n, cost) => +0
(0, 0)

(1, 10) => +10

(2, 15) => +15

(3, 30) => +30

(3, 40) => X (no action)

(3, 50) => X (no action)

cost = 55

(1) Initially enter (0,0) in the PQ. Then take it out. vis[0] = T. Now visit neighbour of 0 with min. cost. So, add (1, 10) (2, 15) (3, 30) in the PQ.

(2) Next take out (1, 10) coz min. cost. Make vis[1] = T. cost = (0+10). Now check unvisited neighbour of 1. So, '0' is already visited. we visit 3. So, (3, 40) added to PQ.

(3) Next take out (2, 15) coz. of min cost. Make vis[2] = T. cost = (10+15). Now, check unvisited neighbour of 2. '0' is already visited. we visit '3' from 2. So, (3, 30) added to PQ.

(4) Now, from PQ take out (3, 30) cz of min. cost. vis[3] = T, cost = (25+30). check unvisited neighbour of 3. All of neighbours of 3 (i.e 1, 2) are already visited.

(5) take out (3, 40) from PQ. As, 3 is already visited, so, no action is further required. we shall not add cost here.

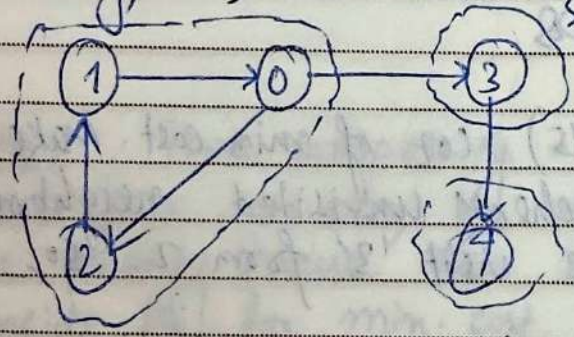
(6) take out (3, 30) from PQ. As, 3 is already visited, so no action is further required. we shall not add cost here.

Now, PQ is empty. we print the cost.

** Strongly Connected Component (scc) **

Scc is a component in which we can reach every vertex of the component from every other vertex in that component.

For e.g.



0 1 2 → scc1

3 → scc2

4 → scc3

To print all such strongly connected component, we shall use Kosaraju's algorithm.

Let's look at 2 more examples —

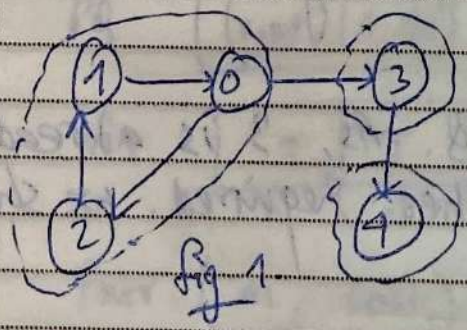


fig 1.

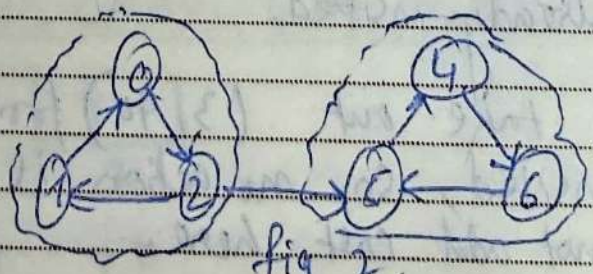


fig 2.

Now, the respective scc's in both the figures are as follows —

fig 1. scc1 → (0 1 2), scc2 → (3), scc3 → (4)

fig 2. scc2 → (0,1,2), scc2 → (4,5,6)

Now, to find out the scc's, we cannot use normal DFS. The reason is that, if we look into both the graphs (fig 1. and fig 2), then by using DFS, we can traverse all the nodes in one go. Therefore, we shall not be able to find out the number of scc's in a graph and the respective component.

Kosaraju's algorithm tell us to modify the DFS algorithm a bit and use reverse DFS in order to find out the scc's.

Reverse DFS :-

on fig 1.

1) Let's start with node (4). From 4 we can't go anywhere. So, it becomes my scc 1.

2) Next, we start with node (3). From 3, we can visit 4, which is it's neighbour, but it's already visited. So, we shall not visit it again. So, it becomes my scc 2.

3) Now, we start with (2). From (2) we can go to (1), which is it's neighbour and unvisited. So, we go to (1). From (1), we can go to (0), which is it's neighbour and unvisited. So, (2) → (1) → (0) becomes my scc. After (0), we can't go anymore, because all of it's neighbours are visited.

So, all the scc's are now visited.

scc 1 → 4

scc 2 → 3

scc 3 → 2 1 0.

Kosaraju's algorithm

Steps :-

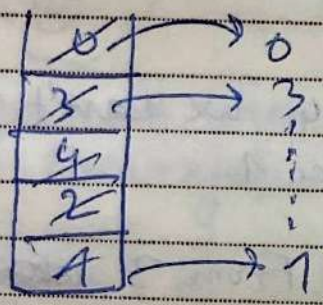
(a) Get nodes in stack (topological sort)

(b) Transpose the graph.

(c) Do DFS according to stack nodes on the transposed graph.

step a)

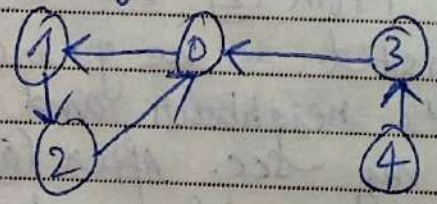
So, for fig 1, we apply topological sort and visit all the nodes and while returning from each node, we put them in a stack.



0 3 4 2 1

step b)

Now, we need to transpose the graph. Basically, it means that reversing the direction of the edges. So, the graph looks like —



step c)

Now, pop out each element from the stack and perform DFS.

So, at first (0) gets popped. We visit (0) and print it. From (0), we can only visit (1). So, (1) is visited and printed. From (1), we can go to (2). (2) is visited and print. Now, from (2), we can go to (0), but it's already visited. So, the first SCC component becomes (0) → (1) → (2).

Next pop out (3). visit (3). From (3), we can visit (0), but it's already visited. So, (3) becomes my 2nd scc.

Next pop out (4). visit (4). From (4), we can visit (3), but it's already visited. So, (4) becomes my 3rd scc.

Next pop out (2). But (2) is already visited. So, we don't perform DFS for it. same for (1).

So, by performing the 3 steps, we got all our scc

Java code :-

```
import java.util.*;
Public class Classroom {
    Static class Edge {
        int src;
        int dest;

    Public Edge (int s, int d) {
        this.src = s;
        this.dest = d;
    }}

Public Static void createGraph (ArrayList<Edge> graph[])
{
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList<Edge> ();
    }
```

```
    graph[0].add(new Edge(s:0, d:2));
    graph[0].add(new Edge(s:0, d:3));
    graph[1].add(new Edge(s:1, d:0));
    graph[2].add(new Edge(s:2, d:1));
    graph[3].add(new Edge(s:3, d:4));
}
Public static void topSort(ArrayList<Edge> graph[],
        int curr, boolean vis[], Stack<Integer> s)
{
    vis[curr] = true;
    for(int i=0; i<graph[curr].size(); i++)
    {
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]) {
            topSort(graph, e.dest, vis, s);
        }
    }
    s.push(curr);
}

Public static void KosarajuAlgo(ArrayList<Edge> graph[],
                                int v) {
// Step 1. - O(V+E)
Stack<Integer> s = new Stack<>();
boolean vis[] = new boolean[v];
for(int i=0; i<v; i++) {
    if(!vis[i]) {
        topSort(graph, i, vis, s);
    }
}
}
```

Action Plan

DEC '22

01 Thu

```
//Step 2 - O(V+E)
ArrayList<Edge> transpose[] = new ArrayList[V];
```

05 Mon

```
for(int i=0; i<graph.length; i++) { vis[i]= false;
    transpose[i] = new ArrayList<Edge>();
}
```

09 Fri

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

13 Tue

```
Edge e = graph[i].get(j);
transpose[e.dest].add(new Edge(e.dest, e.src));
    //e.dest → e.src
}
}
```

17 Sat

```
// Step 3 - O(V+E)
while(!s.isEmpty()) {
```

21 Wed

```
int curr = s.pop();
if(!vis[curr]) {
    dfs(transpose, curr, vis);
```

25 Sun

```
System.out.println();
}
}
```

29 Thu

```
}
```

DEC

01 December
49th Wk • 335-030
Thursday

2022 DECEMBER

M	T	W	T	F	S	S	M	T	W	T	F	S	S
			1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31								

```
Public static void dfs (ArrayList <Edge> graph, int curr,
                        boolean vis[]) {
    vis[curr] = true;
    System.out.print (curr + " ");
    for (int i=0; i<graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        if (!vis[e.dest]) {
            dfs (graph, e.dest, vis);
        }
    }
}

Public static void main (String args[]) {
    int v = 5;
    ArrayList <Edge> graph[] = new ArrayList[v];
    createGraph (graph);
    KosarajuAlgo (graph, v);
}
}
```

Note:- 1) All the steps involved have a time complexity of O(V+E). So, we can say that Kosaraju's algorithm has a time complexity of O(V+E).

2) Scc only applies to directed graph. Reason is, for undirected graph, the whole graph is one scc and no distinct components can be segregated.

** Bridge in Graphs **

Bridge is an edge whose deletion increases the graph's number of connected components. Usually, we study the concept of Bridge in undirected graphs.

Now, to study bridges in graph, we usually use the concept of Tarjan's Algorithm.

E.g.,

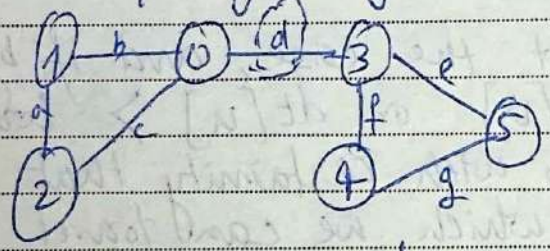


fig 1.

In fig 1. the edges are shown as a, b, c, d, e, f and g. Now, the edge, which if removed, shall increase the component in graphs is called Bridge. Here, the edge 'd' is the bridge.

In Tarjan's algorithm, we majorly use 2 things -

✓ discovery time dt[],

✓ lowest dist. time of all neighbours low[].

* Concept:-

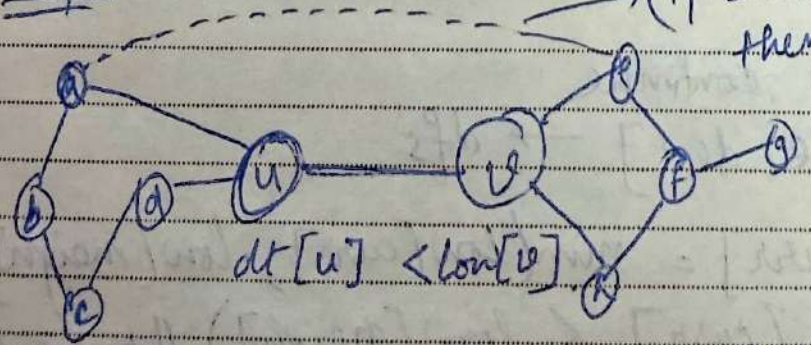


fig 2.

The concept is such that, if dt[u] < low[v], i.e if the discovery time required to reach 'u' is less than the discovery time to reach 'v' or any of it's neighbours, then only we can say, there exists a bridge.

03 December
49th Wk • 337-028
Saturday

2022 DECEMBER

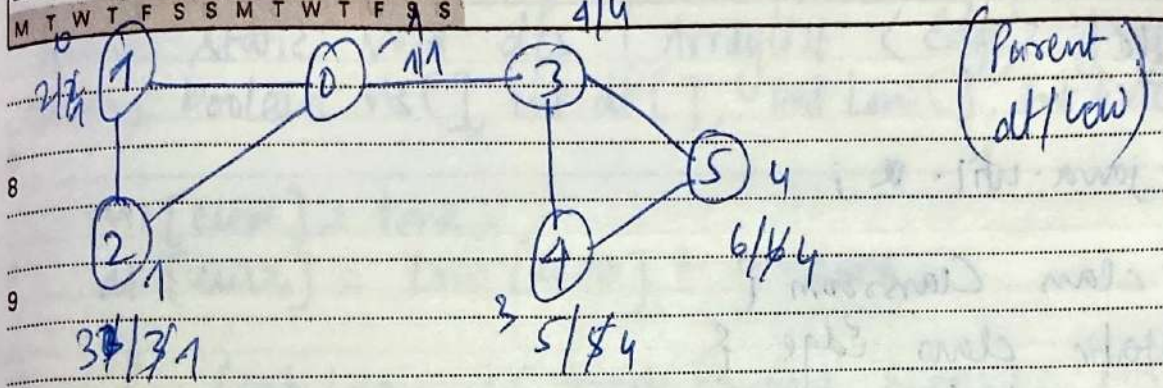
M	T	W	T	F	S	S	M	T	W	T	F	S	S
			1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31								

→ Let's say, we start with vertex 'a'. Now, to jump to another vertex, we need time = 1 (say).
So, if we go from 'a' to 'u' directly, time = 1, or, dt[u] = 1.
Now, if we go to 'v', low[v] = 2 (at least)
So, we can say that the edge between 'u' and 'v' is the bridge, of the graph in fig. 2.

→ If that was not the case, and if by chance, dt[u] = low[v] or dt[u] > low[v], then, we can say with certainity that there exists a back edge via which we can travel to v or it's neighbours. So, then the edge between u & v no longer remain as bridge.

Pseudo code (Tarjan's algorithm) - modified DFS

```
vis[curr] = T
dt[curr] = low[curr] = ++time

for(int i=0 to neighbour)
    Edge e

① Parent → continue
② !vis[e.dest] → dfs
    low[curr] = min(low[curr], low[neigh])
    if(dt[curr] < low[neigh]) // bridge condition
        print(curr + neight)
③ vis[e.dest]
    low[curr] = min(low[curr], dt[neigh])
```



→ visit 0. For '0', Parent = -1. dt[0] = 1. low[0] = 1. visit '1'.

→ For '1', dt[1] = 2, low[1] = 2. Now check neighbours of 1. '0' is parent, so continue. go to 2. For '2', parent = 1, dt[2] = 3, low[2] = 3.

→ Now check neighbours of 2. '1' is parent, so continue. when checked with '0', it's already visited, so update it's own low[]. so, low[2] = min(3, 1) = 1. Now go back to 1.

→ Now, for 1, '0' is parent, so continue. when checked with 2, it's already visited, so nothing to do, except updating low[]. so low[1] = min(2, 1) = 1.

→ Same process followed for (3), (4), (5).

→ Now Returning back from (5), low[5] = min(6,4) = 4. come back to 4. low[4] = min(5, 4) = 4 (following step 2).

→ come back to 3. neighbours of (3) visited. Also, dt[curr] = dt[3] = 4 is not less than low[4] or low[5].

→ come back to 0.
Now, dt[0] = 1 < low[3]. So, bridge exists.
So, we Print 0 → 3. That means that the edge between (0) and (3) is a bridge.

Java code

```
import java.util.*;

public class Classroom {
    static class Edge {
        int src;
        int dest;

        public Edge(int s, int d) {
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        for(int i=0; i<graph.length; i++) {
            graph[i] = new ArrayList<Edge>();
        }
        graph[0].add(new Edge(s:0, d:1));
        graph[0].add(new Edge(s:0, d:2));
        graph[0].add(new Edge(s:0, d:3));
        graph[1].add(new Edge(s:1, d:0));
        graph[1].add(new Edge(s:1, d:2));
        graph[2].add(new Edge(s:2, d:0));
        graph[2].add(new Edge(s:2, d:1));
        graph[3].add(new Edge(s:3, d:0));
        graph[3].add(new Edge(s:3, d:4));
        graph[3].add(new Edge(s:3, d:5));
        graph[4].add(new Edge(s:4, d:3));
        graph[4].add(new Edge(s:4, d:5));
        graph[5].add(new Edge(s:5, d:3));
        graph[5].add(new Edge(s:5, d:4));
    }
}
```

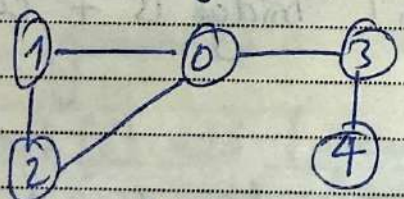
```
Public static void dfs (ArrayList <Edge> graph[], int
curr, boolean vis[], int dt[], int low[], int time, int par){

    vis[curr] = true;
    dt[curr] = low[curr] = ++time;

    for (int i=0; i< graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        if( e.dest == par) {
            continue;
        }
        else if ( !vis[e.dest]) {
            dfs (graph, e.dest, vis, dt, low, time, curr);
            low[curr] = Math.min(low[curr], low[e.dest]);
            if(dt[curr] < low[e.dest]){
                System.out.println("bridge is" + curr + "->" + e.dest);
            }
        }
        else {
            low[curr] = Math.min(low[curr], dt[e.dest]);
        }
    }
}
Public static void getBridge (ArrayList <Edge> graph[], int V) {
    int dt[] = new int[V];
    int low[] = new int[V];
    int time = 0;
    boolean vis[] = new boolean[V];
    for ( int i=0; i<V; i++) {
        if ( !vis[i]) {
            dfs (graph, i, vis, dt, low, time, -1);
        }
    }
}
```

```
Public static void main (String args[]) {
    int v=6;
    ~~ArrayList<Edge> getBridge(ArrayList<Edge>[] graph~~
    ArrayList<Edge> [redacted] graph[] = new ArrayList[v];
    createGraph (graph);
    getBridge (graph, v);
}
}
```

Articulation point

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph.

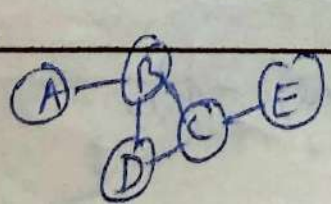
E.g.,



Here, we can say that node (0), (3) can be the articulation points. Because once it breaks, the graph is broken down into more [redacted] components, and all the nodes or vertices are not connected to each other. To realize the articulation point, we shall use the concept of Tarzan's Algorithm.

Ancestor :- a node A that was discovered before our current node in DFS is the ancestor of our current node.

E.g.



So, if we traverse to (E) via (B) → (C), then (B), (C) becomes the ancestor to (E).

In Tarjan's algorithm, we make use of this Ancestor concept. Also, we shall use the discovery time dt[], and lowest discovery time low[] parameters.

Time complexity to find Articulation point using Tarjan's Algorithm is $O(V+E)$.

So, let's talk about the conditions by which we can say that the node is the Articular point (Ap).

✓ Condition 1:- (if node is the starting point, Par = -1).

If node → Par = -1 (start of DFS) & disconnected children > 1, then node is the Articulation Point (Ap).

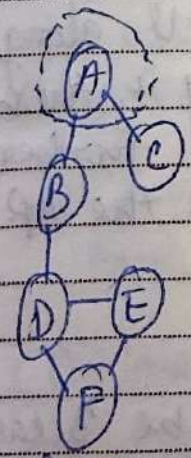


fig. 1.

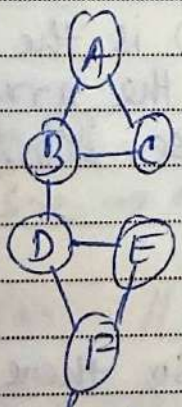


fig 2.

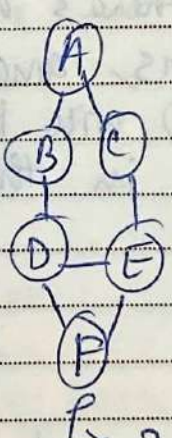


fig. 3.

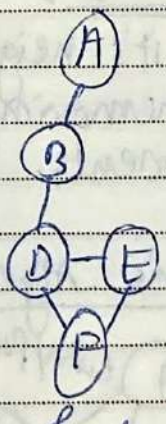


fig 4.

→ For fig. 1, if we start our DFS from node (A), then Par = -1 & disconnected children = 2. So, (A) is the Ap. when we remove node (A), it's edges are also removed. So, we see graph is broken into several components.

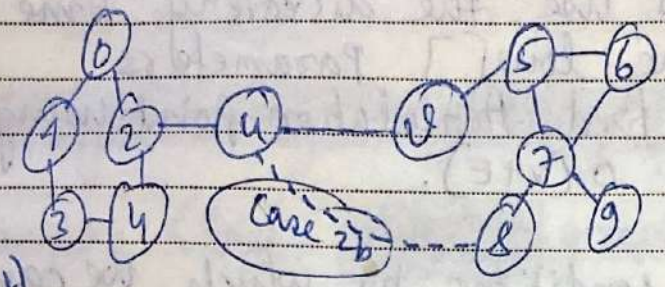
→ For fig 2., Par = -1 but children are not disconnected.

→ For fig 3., children are indirectly connected, so not disconnected.

→ For fig. 4, children = 1. So, not disconnected. So (A) is not Ap in fig 2., 3., and fig 4.

dt[curr] <= (low[neigh] && par != -1), then AP = curr.

Condition 2 :- If node is not the starting point (par != -1)

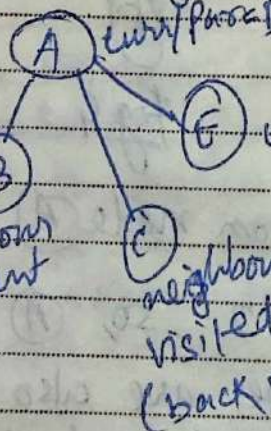


(single root) Case 2a :- Here, we consider the link between (u)→(v) where u is not the starting point of DFS.

Here, there's only a single edge from (u) to reach (v) and it's neighbour. so, we can say (u) is an AP. if there would have been any back edge, then (u) would have not been considered as the AP.

(cycle) Case 2b :- If there's a cycle with u and v, along with it's neighbours, and (u) is the root of it, then also removing (u) will break the graph into further components. so, in this case also, (u) is the AP.

Tarjan's Algorithm



So, there shall be 3 cases-

(I) neighbour is the parent

(II) neighbour is visited, but not parent (back edge)

(III) neighbour is not visited.

→ So, if my neighbour is my parent, then we don't do any action, i.e we continue. In this case, we cannot say with certainty that the current node will be AP.

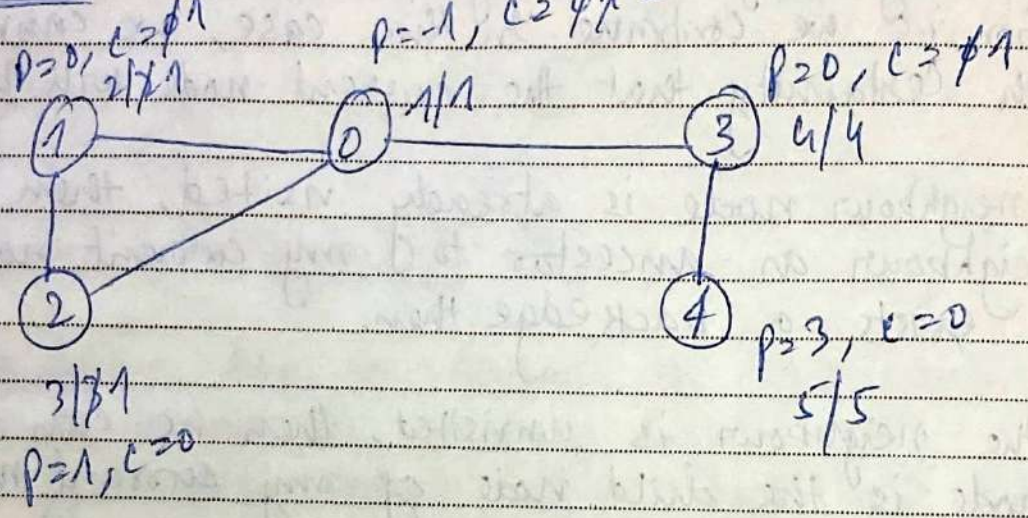
→ If neighbour node is already visited, then we call the neighbour as ancestor to my current node. Of course, there exists a backedge then.

→ If the neighbour is unvisited, then we can say that the node is the child node of my current node.

Pseudo code

```
vis[curr] = true
dt[curr] = low[curr] = ++time
children = 0

for (neighbour) {
  Edge e (src → dest)

(1) neigh = par   // ignore

(2) if (neigh = visited)
    low[curr] = min(low[curr], dt[high])   // explained later

(3) if (neigh = unvisited)
      dfs call
    low[curr] = min(low[curr], low[neigh])
    if (dt[curr] <= low[neigh] && par != -1)
      curr = AP; child++;

if (par = -1 && children > 1)
  curr = AP;
```

Dry run



Java code :-

```
import java.util.*;
public class Classroom {
    static class Edge {
        int src;
        int dest;
    public Edge (int s, int d) {
        this.src = s;
        this.dest = d;
    }
}

Public static void createGraph (ArrayList<Edge> graph[])
{
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList<Edge> ();
    }
```

```
graph [0]. add (new Edge (s:0, d:1));
graph [0]. add (new Edge (s:0, d:2));
graph [0]. add (new Edge (s:0, d:3));

graph [1]. add (new Edge (s:1, d:0));
graph [1]. add (new Edge (s:1, d:2));

graph[2]. add (new Edge (s:2, d:0));
graph[2]. add (new Edge (s:2, d:1));

graph[3]. add (new Edge (s:3, d:0));
graph [3]. add (new Edge (s: 3, d:4));

graph[4]. add (new Edge (s:4, d:3));
}

Public static void dfs(ArrayList<Edge> graph[], int curr,
        int par, int dt[], int low[], boolean vis[], int time,
        boolean ap[]) {

    vis[curr] = true;
    dt[curr] = low[curr] = ++time;
    int children = 0;

    for (int i=0; i< graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        int neigh = e.dest;

        if (par == neigh) {
            continue;
        } else if (vis[neigh]) {
            low[curr] = Math.min(low[curr],
                                 dt[neigh]);
        }
```

```
else {
dfs(graph, neigh, curr, dt, low, vis, time, ap);
low[curr] = Math.min(low[curr], low[neigh]);
if (dt[curr] <= low[neigh] && Par!=-1){
    ap[curr] = true;
}
children ++;
}
}
if (Par==-1 && children>1) {
    ap[curr] = true;
}
}
Public static void getAP(ArrayList<Edge> graph[], int V)
{
int dt[] = new int[V];
int low[] = new int[V];
int time =0;
boolean vis[] = new boolean[V];
boolean ap[] = new boolean[V];
for(int i=0; i<V; i++) {
    if(!vis[i]){
    dfs(graph, i, -1, dt, low, vis, time, ap);
}
}
for(int i=0; i<V; i++) {
  if(ap[i]) {
    System.out.println("Ap:"+i);
}}}
```

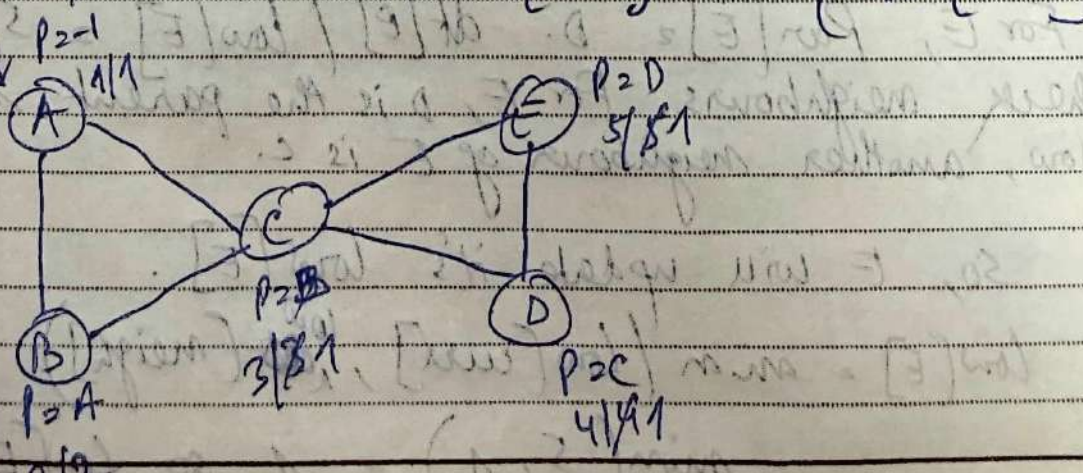
```
public static void main(String args[]) {
    int v = 5;
    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);
    getAP(graph, v);
}
```

(a) note :- Time complexity of Tarjan's Algorithm :- O(V+E), which is better than the naive process of finding articulation point, where time complexity is O(V*(V+E)).

(b) why we are updating the low[curr] = min(low[curr], dt[neigh]) when vis[neigh] is true ?

→ if (vis[neigh] = true) then,
low[curr] = min(low[curr], dt[neigh]).

But let's see why we are doing this. let's say we update this as low[curr] = min(low[curr], low[neigh]).



15 December
51st Wk • 349-016
Thursday

2022 DECEMBER

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Condition of AP :- dt[curr] <= low[neigh]

Let's do a dry run on this graph :—

i) Let's say we start with A. So, P=1, dt[A]/low[A] = 1/1. Now visit neighbour.

ii) go to B. So, for B, Par[B] = A, dt[B]/low[B] = 2/2 Now check for neighbours. For B, A is the Parent, so ignore. For B, C is unvisited. So, go to C

iii) So, for C, Par[C] = B. dt[C]/low[C] = 3/3. Now, C will check it's neighbours. For C, B is the parent, so ignore. For C, A is already visited but not parent.

So, C will update it's low[curr].
So, low[curr] = min(low[curr], low[neigh]) so,

low[3] = min(3, 1) = 1. So, C updates it's low[3] as 1. Now, go to D (unvisited).

iv) For D, Par[D] = C. dt[D]/low[D] = 4/4. Now, from D, C is it's parent, So, ignore. go to E.

v) For E, Par[E] = D. dt[E]/low[E] = 5/5. Now, check neighbours. For E, D is the parent, so ignore. Now, another neighbour of E is C.

So, E will update it's low[E].

low[E] = min(low[curr], low[neigh])

~~= min(5, 1) = 1. So, low[E] = 1.~~

6) Now, control goes back from E to D. Now, D will also update low[D].

low[D] = min[low[curr], low(neigh)]
= min(4, 1) = 1.

Also, D will check the AP condition. As per it,

dt[curr] <= low[neigh]

4 <= 1 → false. So no, AP. Now, go back to BC.

7) For C, dfs is called. low[curr] is updated. So, it will check the condition of AP.

dt[curr] <= low[neigh]

3 <= 1 → false. So, as per the assumed condition, node (C) will not be Articulation point. whereas, if see the graph, (C) is definitely an articulation point.

If we would have updated the condition with

low[curr] = min(low[curr], dt[neigh]),

then For (E), dt[E] / low[E] = 5/3,

For (D), dt[D] / low[D] = 5/4.

Now, at (C), dt[curr] <= low[neigh]

3 <= 3 → true.