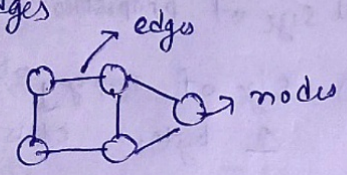
Lecture 84 83
Linked list & its types → Singly, Doubly, circular etc

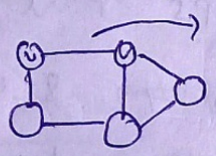
Lecture 85
Intro to graph

① Type of DS which is combination of nodes & edges



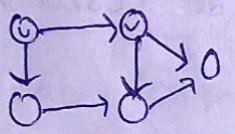
② Type of graph

a) undirected graph



u → v and
v → u
both possible

b) Directed graph



u → v
is possible
but
v → u
not possible

node is an entity in which we can store data.

edge is used for connection of nodes.

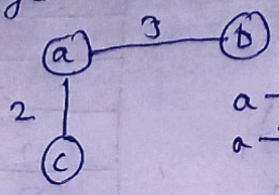
degree is no. of edges connected to a node.

In case of Directed graph we have Indegree and outdegree

Indegree = no. of edges jo mari trf aa rahi hain.

Outdegree = no. of edges jo mere se ja rahi hain.

c) weighted graph
edges has some weight

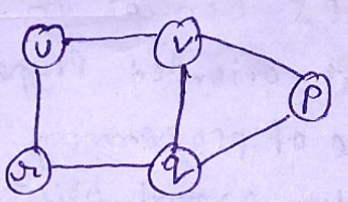


a → b → 3
a → c → 2

If there is no weight than we can assume the weight to be = 1 in undirected graph

weighted undirected graph
weighted directed graph

path

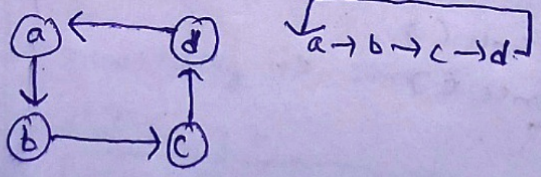


path is sequence of node where each node occur one time

u → v → p ✓
u → r → q → p ✓
u → v → q → r ✓
u → v → q → r → v ✗
(no repeat of node)

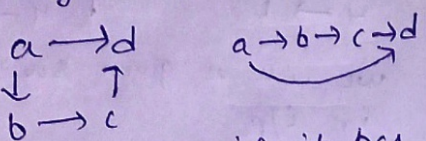
d) Cyclic graph

If while travelling we have reached a node that we have already travelled earlier than it is cyclic graph



Cyclic Directed graph
if it has some weight then its weighted cyclic Directed graph

e) Acyclic graph

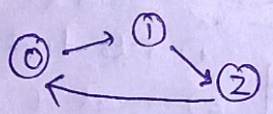


no cycle forms if it has weights, we call it weighted Directed Acyclic graph.

③ Representation of graph
- Adjacency matrix
- Adjacency list

Adjacency matrix

i/p will be like
n = no. of node = 3 (say)
m = no. of edges = 3 (say)
edge list:
0 → 1, 1 → 2, 2 → 0



we will make a 2D matrix

index shows nodes

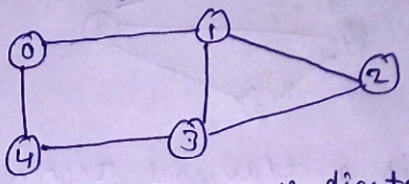
	0	1	2
0	0	1	0
1	0	0	1
2	1	0	0

0 → 1 ✓
1 → 2 ✓
2 → 0 ✓

SC = $O(n^2)$ where
n = no. of nodes.

Adjacency list

n = 3, m = 3
0 → 1, 1 → 2, 2 → 0
we list all nodes and show their connections
Let say our graph is



Above one is a undirected graph so
0 → 1, 1 → 0
0 → 4, 4 → 0
1 → 3, 3 → 1
3 → 2, 2 → 3
4 → 3, 3 → 4
1 → 2, 2 → 1
all are true so we make list like

0 → 1, 4
1 → 0, 2, 3
2 → 1, 3
3 → 1, 2, 4
4 → 0, 3

This was our adjacency list we implement it using

Single object (LHS) → list (RHS)

map < int, list < int >> ;

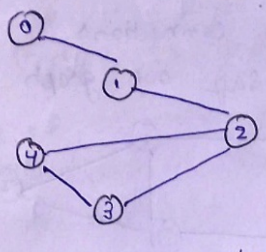
or

vector < vector < int >>

Lecture 86

BFS traversal in graph

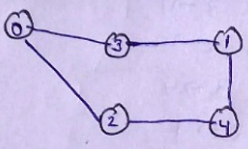
Breadth first search



This is a traversal technique

o/p → 0, 1, 2, 3, 4

already travelled nodes will not be printed again



o/p → 0, 3, 2, 4, 1

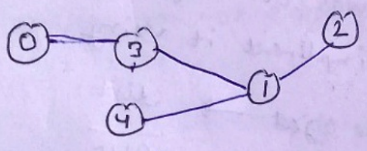
1) we need a DS to track a node is visited or not.

DS = Data Structure

2) let say we take

unordered_map<node, bool>

3) we need a Queue to insert node & process its neighbours in FIFO order.



adj list

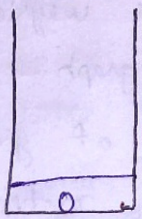
adj list

0 → 3
1 → 2, 3, 4
2 → 1
3 → 0, 1
4 → 1

visited

0 → False
1 → F
2 → F
3 → F
4 → F

we will be given source node in i/p



1)
if(!visited[node])
{ // apply BFS
}

2) Take out front node from Queue, mark visited. 0, True

3) pop front node from queue

4) ans = print 0

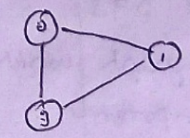
5) check neighbours of front node and push neighbour in Queue say 3

6) Repeat step

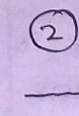
1) Take frontnode = 3
2) mark it visited
3) pop from Queue
4) print 3
5) check its neighbours 0, 1 but we can't put 0 in Queue as visited[0] = true but push 1 in Queue

6) Repeat steps above

At the end our queue becomes empty, we can have graph like



Component



Component

Above 2 are not seperate graphs but are Component of a graph we can call it disconnected graph. To Tackle it we can Run a for loop at starting

```
for (i=0 → i<n)
{ if (!visited [node])
   { bfs ();
   }
}
```

all 6 steps here

★ if we need answer (print) in sorted order then in adjacency list we use set instead of list in

unor_map <int, set<int>> adj

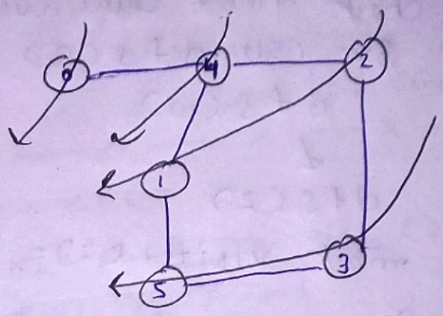
// TC = O(N + E)
↓ ↓
Node edges

// SC = O(N + E)

Lecture 87

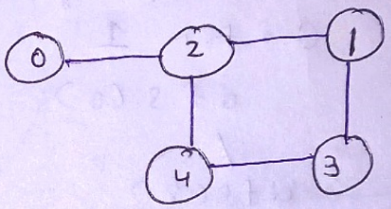
DFS in Graph

(Depth First Search)



BFS: 0, 4, 2, 1, 3, 5

DFS: 0, 4, 2, 3, 5, 1



we can have a disconnected graph also so

```
for (i=0 → i<n)
{ if (!visited [i])
   { dfs ()
   }
}
```

we will have a visited map

unor_map <int, bool> visited;

Approach to DFS

we will make adjacency list

0 → 2
1 → 2, 3
2 → 0, 1, 4
3 → 1, 4
4 → 2, 3

1) We call dfs (0)

2) We mark visited [0] = 1

3) Check next level we see, 2
So call dfs(2)
dfs(0)
↓
dfs(2)
mark visited [2] = 1

4) Go next level of 2
i.e 0, 1
but 0 already visited
Go for 1
dfs(0)
/
dfs(2)
/
dfs(1) mark visited [1] = 1

We check next level using adjacency list.

5) for dfs (3)

(0) 0
↓ ↑
(2) 2
↓ ↑
(1) 1
/ ↑
(3) 3
/ ↑
(4) 4

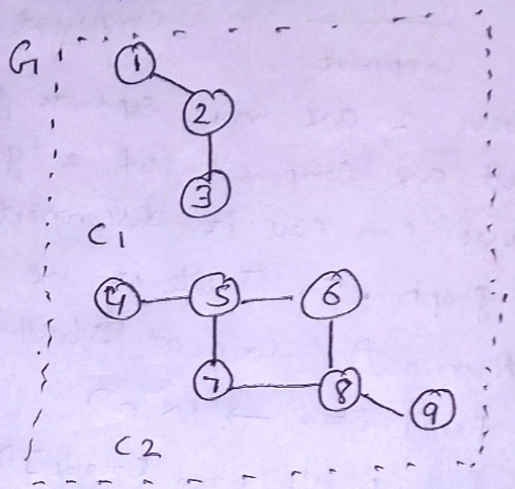
ans = 0, 2, 1, 3, 4

Come back once graph all nodes are visited

Lecture 88

Cycle Detection in Undirected graphs using BFS and DFS

There is a graph which has multiple component

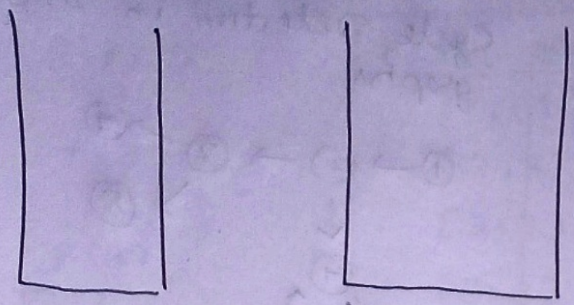


Adj list

1 → 2
2 → 1, 3
3 → 2
4 → 5
5 → 4, 6, 7
6 → 5, 8
7 → 5, 8
8 → 6, 7, 9
9 → 8

We will use BFS

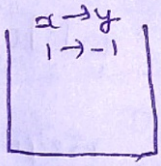
So we need visited DS and one Queue



Queue Visited

1) we move to 1 (no cycle)
2) we move to 2
3) ,, 3
4) ,, 4
5) ,, 5
6) ,, 6, 7, 8
7) we move to 6, 7, 9 from 8 and see 8 is already visited means there is a cycle.

we maintain a DS to store parent



1 ka no parent so Let it be -1

1 → -1

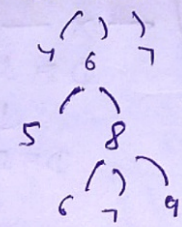
add 1 in Queue, pop it.
now check neighbour of 1 i.e 2 so mark 2 visited
store 2→1 in parent
Add it in Queue
Go to neighbour of 2 i.e 1, 3
1 is already visited and parent of 2 so put 3
3 ke neighbour 2 but its already visited.

— x — x — x —

To neglect a node
1) its visited = true SP
2) its a parent of someone

— x — x — x — x —

now src = 4
4 ka parent nahi Hai so
4 → -1 in parent
visited [4] = True
go to 5, 5 → 4



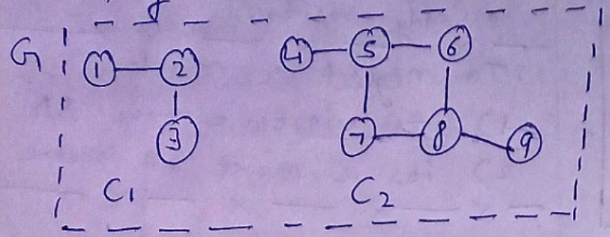
~~already present so there is a cycle.~~

— x — x — x — x —

How to detect cycle
1) node is already visited
2) but it is not parent of anybody then there is Cycle present.

— x — x — x — x —

Using DFS

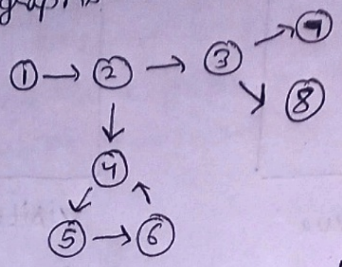


parent	visited
1 → -1	1 → T
2 → 1	2 → T
3 → 2	3 → T

DFS(1,-1)
↓
DFS(2,1)
↓
DFS(3,2)
— x — x — x —
DFS(4,-1)
↓
DFS(5,4)
↓
DFS(6,5)
↓
DFS(8,6)
↓
DFS(9,8)

DFS(7,8)
↓
DFS(5,7)
(5 is visited but not parent so cycle is present)

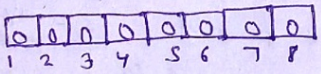
Lecture 89
Cycle Detection in Directed graphs.



Detect cycle in graph above

Adj list
1 → 2
2 → 3, 4
3 → 7, 8
4 → 5
5 → 6
6 → 4
7 →
8 → 7

visited



Keeping in mind disconnected graph

```
for(i=1 → n)
{  if(!vis[i])
   {  dfs()
   }
}
```

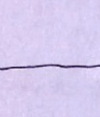
Using DFS

f(1) → f(2) → f(3) → f(7) → ∝ , f(8) → f(7) (already visited)

Parent DS

[1, 2]
[2, 3]
[3, 7]
[3, 8]

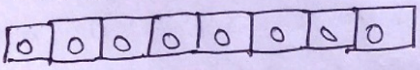
Here we see 7 is not a parent but is already visited means there is cycle present but there is no cycle in



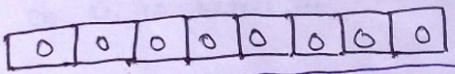
So we cannot use logic of previous lecture here

we need to make another DS (DFS visited) to track DFS visit of each node

visited



DFS visited



f(1) → DFS vis [1] = true
f(2) → DFS vis [2] = true
f(3) → DFS vis [3] = true

This is how DFS visited keep track of DFS visit of each node

When we return back in recursion tree

f(3) → f(7) → f(3) we mark DFS vis [7] = false because

Working of algo

f(1) [mark vis[1] and DFS[1] = true]
↓
f(2) [mark vis[2] and DFS[2] = true]
↓
f(3)
↓
f(7)
↓
∝ (no adjacent node so go back and mark DFS[7] = false]

f(3) → f(4) [,,] → f(5) [,,] → f(6) [,,] [return from f(3) and DFS[3] = 0] [mark DFS[7] = false]

f(8) [,,]
↓
f(7) is already visited so no call of f(7) go back from f(8) and mark DFS[8] = 0

ZOOM f(2)

f(2) → f(3)
f(2) → f(4) [,,] → f(5) [,,] → f(6) [,,] → f(4) but vis[4] & DFS[4] already true so cycle is Detected in graph

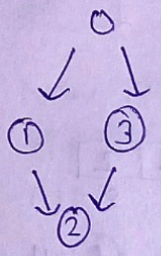
Lecture 90

Topological Sort using DFS

DAG → Directed Acyclic graph

Top sort can be only applied in DAG.

It is a linear ordering of vertices such that for every edge b/w u → v u always exist before v.



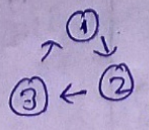
0 → 1, 3
1 → 2
2 →
3 → 2

0 → 1, 3
means 0 is before 1, 3 ?
Yes
1 → 2
1 is before 2 ?
Yes
3 → 2
3 is before 2

So 0 1 3 2 is valid Top sort
but 3 2 1 0 is invalid ∝
as 0 is before 1 not 1 before 0.

It applies in all edges.

Why only DAG ?



1 → 2
2 → 3
3 → 1

Here 1 is before 2, 2 is before 3 but 3 is before 1 so 1, 2, 3 not valid Topo Sort so i.e why Topo sort only apply in DAG.

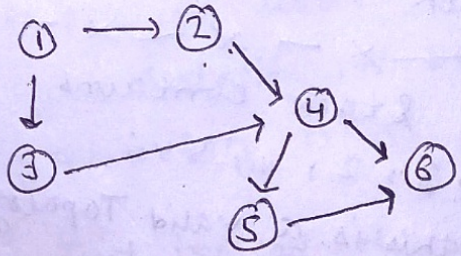
TO Detect whether Cycle present in Directed Graph we can use Topological Sort

Using DFS

we use visited DS and a stack [LIFO]

To consider Dis connected graph case we use

```
for (i = 1 → n)
{ if (!vis[i])
    { DFS()
    }
}
```



1 → 2, 3
2 → 4
3 → 4
4 → 5, 6
5 → 6
6 →

visited

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

1
3
2
4
5
6

main()
dfs(1) push(1)
[mark vis[1]]
true
push(2)
dfs(2) [,,]
push(4)
dfs(4) [,,]
push(5)
dfs(5) [,,]
push(6)
dfs(6) [,,]
(no node)
Go back and push it in stack while going back

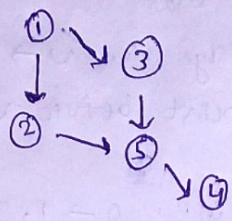
push(3)
dfs(3) [,,]
dfs(4) ✗
dfs(6) ✗

— × — × — × — × — × —

Our stack contains
1, 3, 2, 4, 5, 6
and this is a valid Topological Sort

why use stack
As stack is LIFO so we push that element in stack which can't be propagated further and we know that will be popped at last as LIFO

Lecture 91
Topological Sort using Kahn's algorithm
using BFS
we will use a Queue and an Array which stores indegree of each node.



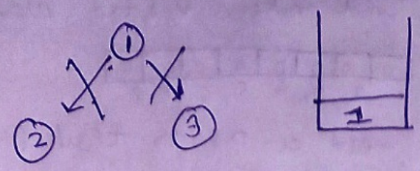
indegree

0	1	1	2	2
1	2	3	4	5

① Find indegree of all nodes
② queue → insert all nodes with 0 indegree
③ Do BFS
Get front element from queue
pop it from Queue
store it in answer
Find neighbour of front

adj list
1 → 2, 3
2 → 5
3 → 5
5 → 4
4 →

Store 1 in answer
ans = {1}

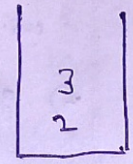


As we pop 1 means ② & ③ gets cancelled and indegree [2] and [3] reduces by 1

earlier indeg[2] = ✗ 0

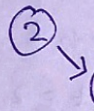
indeg[3] = ✗ 0

So push them in Queue as their indegree is 0 now



2) Now perform same opp for 2 as FIFO

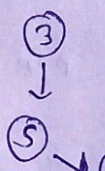
store in ans
ans = [1, 2]



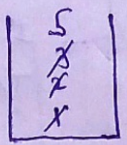
indeg[5] = ✗ 1 ≠ 0 so don't push in Queue

3) Now front node = 3

store in ans
ans = [1, 2, 3]

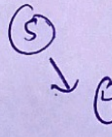


indeg[5] = ✗ 0
Push it in Queue



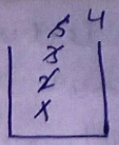
4) Front = 5

store in ans
ans = [1, 2, 3, 5]



indeg[4] = ✗ 0
Push in Queue

5) front = 4



neighbour of 4 = not found

store in ans
ans = [1, 2, 3, 5, 4]

So processing is Done

answer = [1, 2, 3, 5, 4]

This is valid Topo Sort

— ✗ — ✗ — ✗ — ✗ —

Lecture 92

Cycle Detection in Directed graph using BFS

1) We know we can apply Topological Sort only in DAG. if its a cyclic graph then TOPO Sort is ~~not~~ invalid.

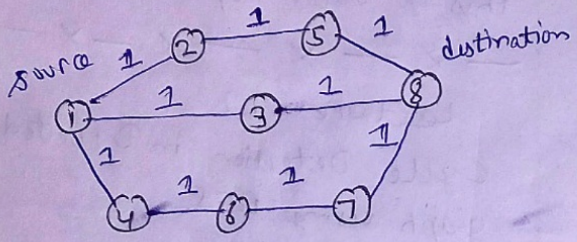
2) In TOPO Sort using Kahn algo we use BFS

3) we can use same code as previous lecture for Cycle Detection in Directed graph using BFS.

4) Just instead of ans, use cnt variable and at end if cnt = no. of vertices then its Acyclic and valid TOPO order, cycle not present else (cnt != n) so cycle is present

Lecture 93

Shortest path in undirected graph we have a undirected graph we have a source node and a destination node. we need to find shortest path b/w source to destination.



path1 = 1 → 2 → 5 → 8
path2 = 1 → 3 → 8
path3 = 1 → 4 → 6 → 7 → 8

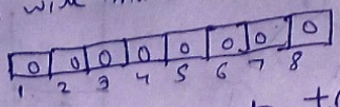
Let say weight of each edge
path1 has 3 edge
path2 has 2 edge
path3 has 4 edge
so path2 is shortest

Algorithm

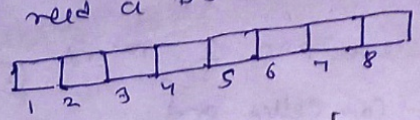
we can use BFS

adj list
1 → 2 → 3 → 4
2 → 1, 5
3 → 1, 8
4 → 1, 6
5 → 2, 8
6 → 4, 7
7 → 6, 8
8 → 3, 5, 7

we will make a visited array

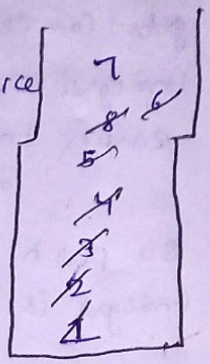


we need a DS to track parents



we need a Queue

1) initially push source node in Queue
mark it visited
mark its parent -1 as it has no parent



2) get front node
go to neighbour
insert each neighbour in Queue, mark them visited, mark their parent.

3) Get new front = 2
we get 1, 5
1 is already visited
Take 5, push mark visited, parent.

4) newfront = 3, pop it
neighbour = 1, 8
1 already Done
Take 8.

5) Front = 4
1 6

6) Front = 5
2 8

7) Front = 8
3 5 7

8) Front = 6
2
4 7
∝ ∝

9) Front = 7
6 8
∝ ∝

now Queue is empty and we have our whole parent array ready

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

BFS always go level by level using shortest path only.

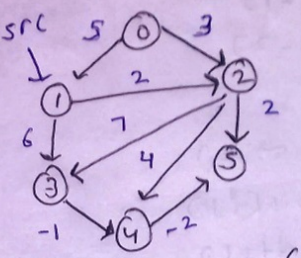
we know
source = 1
destination = 8

To reach 8, we need to reach its parent i.e 3
To reach 3, we need to reach its parent i.e 1
To reach 1, its parent = -1

8 ← 3 ← 1 is path but its reverse so

Reverse it
answer is 1 → 3 → 8

Lecture 94
Shortest path in DAG
Directed Acyclic graph



Let src node = 1

1 → 0 → ∞ (no directed path)
1 → 1 → 0
1 → 2 → 2
1 → 3 → 6 i.e 6 or 2+7 = 9
1 → 4 → 5 i.e 2+4 or 6-1 = 5
1 → 5 → 3 i.e 2+2 or 6-1-2 = 3

shortest path if src node = 1 :
{-∞, 0, 2, 6, 5, 3}

Approach

1) Topological sort is for DAG we know.
It gives you which node lie before which node
2) utilise Topo sort
3) update distance array using it.

we need a stack, vis array for Topo Sort
adj list will be little different as this is a weighted graph so we need to store [node value, weight of node]
unordered_map <int, list <pair <int, int >>>
earlier it was
unordered_map <int, list <int >>

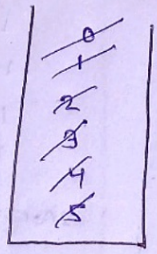
0 → [1, 5], [2, 3]
1 → [2, 2], [3, 6]
2 → [3, 7], [4, 4], [5, 2]
3 → [4, -1]
4 → [5, -2]
5 →

```
for (i → 0 → n)
 { if(!vis[i])
    {  dfs()
    }
 }
```

main()
dfs(0)
dfs(1)
dfs(2)
dfs(3)
dfs(4)
dfs(5)

(has 4, 5 but already in stack so return)

while returning store it in stack



Topo order is coming out to be
0, 1, 2, 3, 4, 5

★ now make distance array to store distance and shortest path initialised with some max value INT_MAX

	0	1	2	3	4	5
distance array	∞	∞	∞	∞	∞	∞
			0	2	6	

dist[src] = 0

get st.top() → 0

1) 0 has ∞ dist from 1 leave
2) 1 has 0 dist with 1 and can go to 2 and 3 from 1
3) 2 leaves Now

(0) (1) —2→ (2)
(1) —6→ (3)

So update dist[2] = 0+2 = 2
dist[3] = 0+6 = 6 in array

(2) Now st.top = 2 we can go to 3, 4, 5 from 2

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

2 → 3 = 2+7 = 9 ✗ 6
we need short path so neglect

2 → 4 = 2+4 = 6 < ∞
update it in dist array
dist[4] = 6

2 → 5 = 2+2 = 4 < ∞
update, dist[5] = 4

Dist Array become

	0	1	2	3	4	5
	∞	0	2	6	6	4

(3) Now st.top = 3

(6) 3 —−1→ 4

3 → 4 = 6−1 = 5 < 6
update it, dist[4] = 5

(4) st.top = 4

(5) 4 —−2→ 5 = 5−2 = 3 < 4
dist[5] = 3

(6) st.top = 5

5 → ? (no place so stop)

Stack become empty

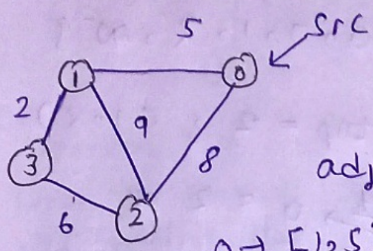
Our dist array become

{∞, 0, ?, 6, 5, 3}

— x — x — x — x — x —

Lecture 95

Dijkstra's shortest path



adj list

0 → [1,5], [2,8]

1 → [0,5], [2,9], [3,2]

2 → [0,8], [1,9], [3,6]

3 → [1,2], [2,6]

0 → 1 = 5

0 → 2 = 8

0 → 3 = 7

0 → 0 = 0

= [0, 5, 8, 7]

1) Dist array is need

∞, ∞, ∞, ∞

0 1 2 3

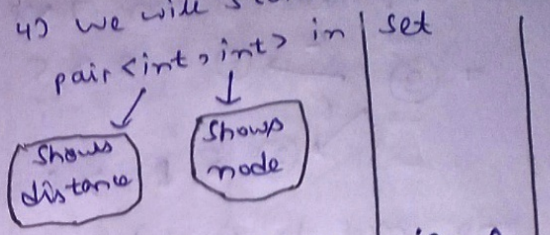
2) we need another DS which can give us shortest node so either use min heap priority queue

or

use set

3) we will use set

4) we will store pair<int, int> in set



5) Initialise set with (0, 0) for node 0 considering source = 0

6) get topnode = 0 and pop it go to neighbour of topnode we can go to 1, 2 from 0

0 → 1 = 0 + 5 = 5 < ∞

update dist[1] = 5

0 → 8 = 0 + 8 = 8 < ∞

update dist[2] = 8

7) store them in set while update

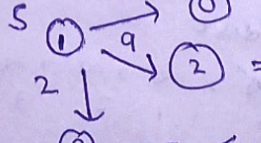
(7, 3)
(8, 2)
(5, 1)
(0, 0)

8) get topnode

As it is a set, not a Queue.

Topnode will always be the one with shortest distance.

topnode = 1 . pop it

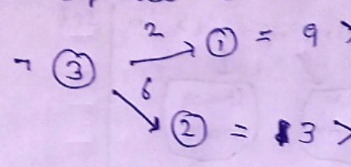


0 = 10

2 = 14 > 8

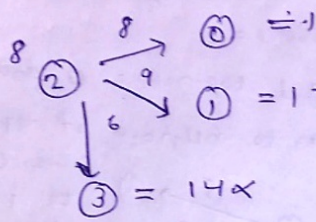
3 = 7 < ∞ update and store in set

9) top node = 3 as 7 < 8 distance



no need to update

10) get top node = 2

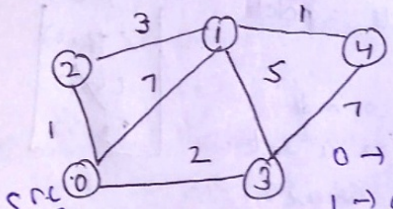


no need to update

Our distance array looks like

[0, 5, 8, 7] is our answer

Another example



adj list

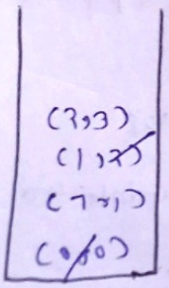
0 → [1,7], [2,1] [3,2]

1 → [0,7] [2,3] [3,5] [4,1]

2 → [0,1], [1,3] ~~[3,2]~~

3 → [0,2] [1,5] [4,7]

4 → [1,1] [3,7]



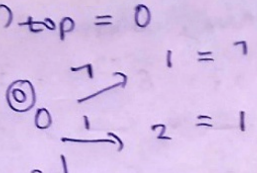
Set
(dist, node)

dist

0 ~~∞~~, ∞, ∞, ∞, ∞
0 1 2 3 4

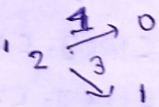
dist[src] = 0

1) top = 0



{0, 7, 1, 2, ~~∞~~}

2) top = 2, (1, 2)



{0, 4, 1, 2, ∞}

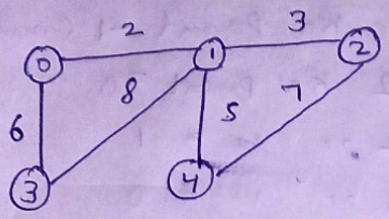
now we have two entries for node 1 in set (4, 1) & (7, 1) we need to store latest with short distance so update (7, 1) to (4, 1) in set

3) Rest whole algo works in same as previous example

Lecture 96

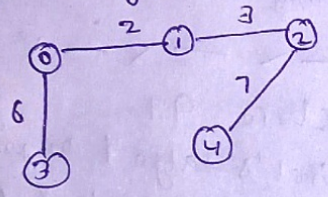
Minimum spanning Tree

Prim's Algorithm



Spanning Tree

when you convert into a Tree such that it has n nodes and n-1 edges such that every node is reachable by any other node means there is no cycle as there is no cycle in Tree.



Min Spanning tree means minimum cost

To get min Spanning Tree we can use prim's algo or Kruskal algorithm

Prim's algo

we need 3 DS for it.

Key array to store key where array index shows node value

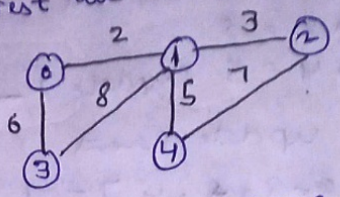
mst array to store min Spanning tree initialised with false

parent array to store parent of a node ini with -1

Let Source = 0 node so

Key [0] = 0

rest all initialised with ∞



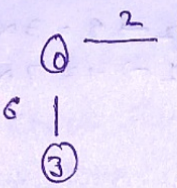
1) mark Key [0] = 0

parent [0] = -1

get min value from Key array i.e 0 and mark it True in mst

mst [0] = T

Get all adjacent node of 0



update Key [1] = 2, Key [3] = 6

mark parents

parent [1] = 0

parent [3] = 0

2) Repeat Process

Get min in Key but keep in mind ~~not~~ that

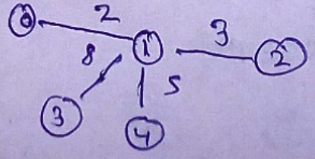
mst [index] = false

so we can't take 0 now

min (Key) = 2 i.e for node value 1.

parent [1] = -1

mst [1] = True



check values for

1 → 0 = 0 ✗
1 → 2 = 3 (update)
1 → 3 = 8
1 → 4 = 5 (update)

p and update parent

parent [? , 4] = 1

Key [2] = 3 , Key [4] = 5

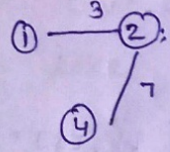
(We will update parent only for updated nodes)

3) Get min from key key in mind mst [ind] = False

node = 2

mst [2] = T

parent [2] = -1

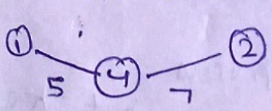


2 → 2 = 3 ✗
2 → 4 = 7 ✗

4) Get min from key

ind = 4

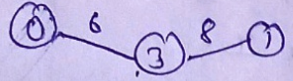
mst [4] = T



(no update)

5) ind = 3

mst [3] = True



(no update)

now all key [ind] = True so execution stops

6) write parent array

-1 , 0 , 1 , 0 , 1

0 1 2 3 4

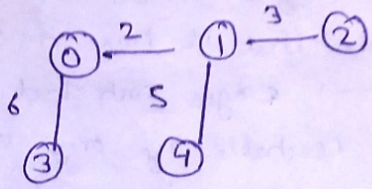
0 ka parent -1 (none)

1 ka parent = 0

2 ,, ,, = 1

3 ,, ,, = 0

4 ,, ,, = 1



This is our minimum spanning Tree.

— x — x — x — x —

Lecture 97

Kruskal's algo | Disjoint Set | Union by rank & path compression.

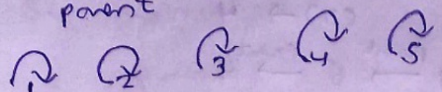
Disjoint Set (DS)

We use this Disjoint set to do Kruskal's algo.

1) Let say we have 2 nodes u and v. using DS we can check whether these 2 nodes, u and v belong to same component or Different Component of a graph.

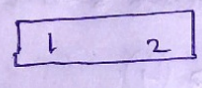
2) There are 2 important operations
a) FindParent or FindSet ()
b) Union () or UnionSet ()

3) Let say we have 5 disconnected component where each node is its parent



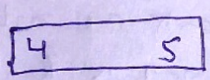
Find Parent (1) → 1
(2) → 2
⋮
(5) → 5

Union (1, 2)
we make single comp containing 1 and 2 where 1 is parent



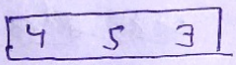
So now there is only one component for 1 & 2 instead of 2 seprate components

Same way
Union (4, 5)



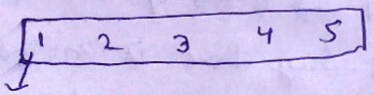
is a single component now where 4 is parent

Union (3, 5)
now



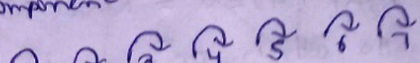
is a component which is single where 4 is parent

Union (1, 3)



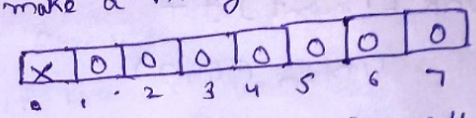
Union by Rank & path Compression

Let say we have 7 disconnected component



find Parent (1) → 1
(2) → 2
⋮
(7) → 7

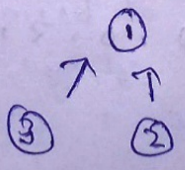
To find union (1, 2) we make a Array of Rank



Initially Rank = 0 for all and we start from 1 so 0 = x

1) FindParent (1) and (2)
2) Check Rank of 1 and 2
if Rank(1) = Rank(2)
we can attach anybody with anybody so
parent [2] = 1
Rank [1] ++

Union (2, 3)
1) Find Parent (2) → 1
and (3) → 3
2) Rank (1) = 1
Rank (3) = 0
Rank (1) > Rank (3)
so parent [3] = 1

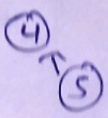


Union (4, 5)

1) Parent(4) → 4
   (5) → 5

2) Rank (4) = 0 = Rank(5)
   So attach anyone with anyone

3) Parent [5] = 4
   rank [4] ++



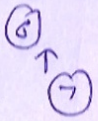
So when Rank is same we mark anyone as parent and do rank [Parent] ++

Union (6, 7)

1) Parent (6) → 6
   (7) → 7

2) Rank = same = 0 for 6, 7

3) Parent [7] = 6
   Rank (6) ++

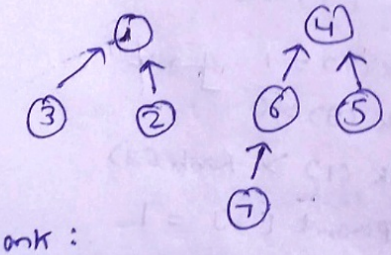


Union (5, 6)

1) Parent (5) → 4
   (6) → 6

2) Rank (4) = 1
   (6) = 1

3) Parent [6] = 4
   Rank [4] ++



Rank :

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

Union (3, 7)

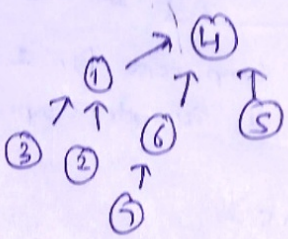
1) Parent (3) → 1
   (7) → 4

7 —Parent→ 6 —Parent→ 4

So Parent (7) ≠ 6 = 4

2) Rank (1) = 1
   Rank (4) = 2

3) Rank (1) < Rank (4)
   So Parent [1] = 4



When Rank are not equal And Rank 1 < Rank 2 then Parent [Rank 1] = Rank 2

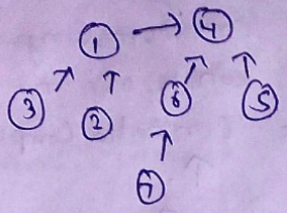
To find Parent (7)
we travelled a lot like
7 → 6 → 4
Let say there is another node (8) then we travel
8 → 7 → 6 → 4

So To optimise this time consuming thing, we use path compression

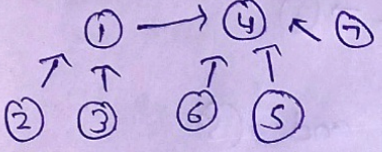
When we know

parent[7] = 6 and parent[6] = 4

So instead of



Do this



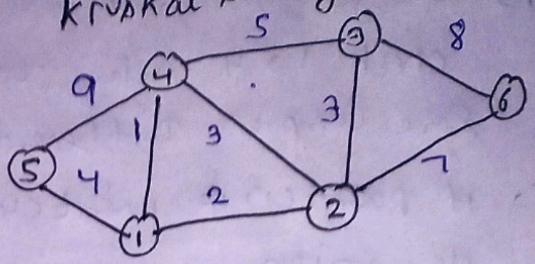
parent[7] = 4 now directly

So this is path compression logic

while doing union, we always try to merge a short Tree under a long tree so that depth do not increase whereas if we merge long tree under a short tree, length/depth of Tree ↑ which is not optimised way.

Above was logic for Disjoint Set now we see kruskal's algo.

kruskal's algo



To check whether 3, 4 lie in same component we check parent(3) and parent(4)

if both are =, they lie in same component

if both are ≠, they lie in Diffent component

we need a

1) adj list ✗ (no need)
2) we need a linear DS Array to store u, v, weight in sorted order

wt, u, v	Sorted
1 → 2, 2 ✗	wt u v
1 → 4, 1 ✗	1 1 4
1 → 5, 4 ✗	2 1 2
4 → 5, 9 ✗	3 2 3
4 → 3, 5 ✗	3 2 4
2 → 4, 3 ✗	4 1 5
2 → 3, 3 ✓	5 3 4
2 → 6, 7 ✗	7 2 6
3 → 6, 8 ✗	8 3 6
	9 4 5

we need to sort in order of weight, u and v can be in any order

1) we get 1, 1, 4

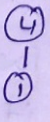
Check 1, 4 belong to same comp or Different

if parent(1) ≠ parent(4)

do union

if parent(1) = Parent(4)

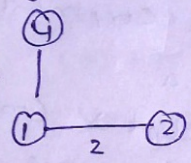
Do nothing



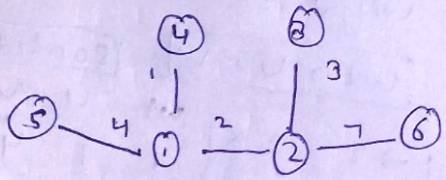
2) 2, 1, 2

Check parent(1), parent(2)

merge as parent not equal



3) Do this for all element from sorted list



This is our minimum spanning tree.

// TC

Sorting = $m \log m$, m = edges

Find parent or union takes constant time

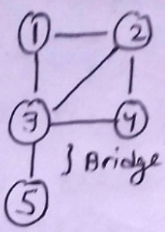
// TC = $O(m \log m)$

// SC = $O(n) + O(n)$

$\simeq O(n)$

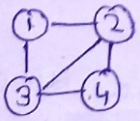
Lecture 98

Bridges in Graph

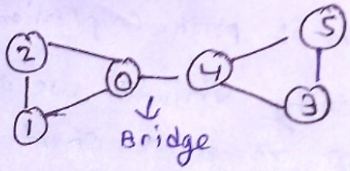


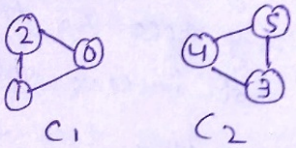
Bridge is an edge on removing which our no. of Connected Component ↑.

on removing (5) Comp ↑ from 1 → (2)



and (5)



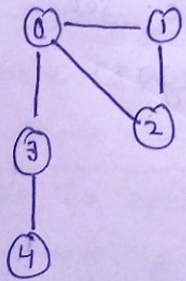


C1 C2

Brute force

Pick every edge and apply DFS to check no. of Connected Components.

Optimised Approach



We need 4 DS here

Discovery: Keep timer of reaching a node (-1)

Low: Keep earliest time of reaching a node (all -1 initially)

Parent: who is parent of node (-1)

visited: check visited node (-1)
We keep a timer = 0

1) Initially Dis[0] and Low(0) = 0 = 0

vis[0] = True, time = 0

2) move to 0 → 1
Dis[1] = 1
Low[1] = 1
vis[1] = T
Parent[1] = 0
timer = 1

3) 1 → 2
Dis[2] = 2
Low[2] = 2
vis[2] = T
Parent[2] = 1 ; timer = 2

4) 2 → 0 but vis[0] = T already so its a back edge means we could have gone from 0 to 2 Directly so Dis[2] remains same but Low[2] can be updated

low[node] = min(low[node], Dis[neighbour])

Whenever we face a back edge case we update low[node]
low[node] = min(low[node], Dis[neighbour])

Low[2] = 0 now timer = 3

Dis

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

Low

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

Parent

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

Vis

T	T	T		
~~F~~	~~F~~	~~F~~	F	F
0	1	2	3	4

5) we can go 2 → 1 also but if (neighbour == parent)
{ ignore ;
}
because we come from 1 → 2 so parent[2] = 1 so ignore this case and return
But while returning make some update in parent node i.e

low[parent[node]] =
min(low(parent[node]), low(child));

So low[1] = 0 as low[2] is updated to 1

we need to bridge also
How to check bridge?

if (low[neighbour] > disc[node])
{
  // Bridge is present
}

6) Same way we move to 3

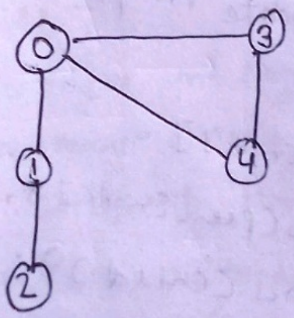
2 → 1 → 0 and check bridge for this edge.
3
4

— x — x — x — x ——

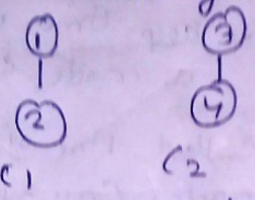
Lecture 99

Articulation points in Graph | Tarjan's algo

It is a node on removing which our graph divides into 2 or more components

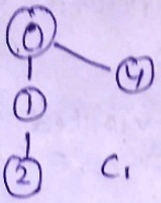


On removing 0



C1 C2

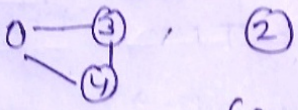
0 is one articulation points



C1

3 is not a Art point

On removing 1



C1 C2

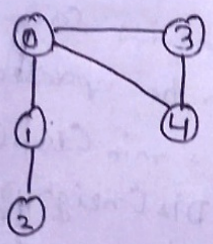
1 is a articulation point

(codes-tudio doc link on code)

Approach

1) Brute force
Remove all vertices and check whether it makes graph disconnected or not.

2) OPTIMISED APPROACH



We will use some DS
Discovery, low, parent,
visited all initialised with
-1 & F.
int timer = 0

To check whether a node is articulation point or not?

```
if (low[neighbour] >= disc[node]
      && parent != -1)
{
    // It is a Articulation
       point
}
```

Algo Begins

① timer = 0
vis[0] = T
dis[0] = 0
low[0] = 0

② timer = 0 1
we are at node = 3
vis[3] = T
dis[3] = 1
low[3] = 1
parent[3] = 0

③ timer = 0 1 2
at node = 4
vis[4] = T
dis[4] = 2
low[4] = 2
parent[4] = 3

④ timer = 2 3
node 4 → 0 and we see it is a back edge
So update low[4] = 0

low[node] = min(low[node], disc[neighbour])

Keeping in mind (neighbour != parent)

⑤ update low of parent while returning from Back edge.
low[3] = 1 0

low[3] = min(low[3], low[4])

⑥ check whether 3 is articulation point or not

low[4] >= dis[3]
// false
So 3 is not a articulation point.

⑦ Now make similar calls for
0 and 1 and 2

(8) To find Art point we use condn

(low[nghr] >= disc[node && parent != -1)

parent ≠ -1 means it should not be a root node as parent[root node] = -1 always

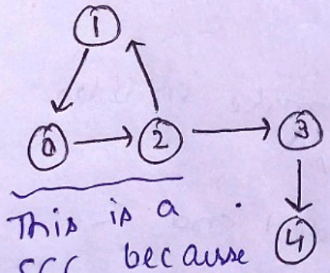
and child > 1 also a necessary condn for a node to be a articulation point.

— X — X — X — X —

Lecture 100

Kosaraju's Algo for Strongly connected Components (SCC)

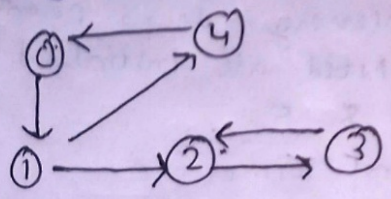
It is used to find / count strongly connected components in graph.



This is a SCC because does not matter we start from any node, we will cover all nodes in the end after travelling

This is not a SCC as we can't go to anywhere from 4

SCC_3 = 3, 4, 0 2 1



0 4 1 ⇒ SCC1

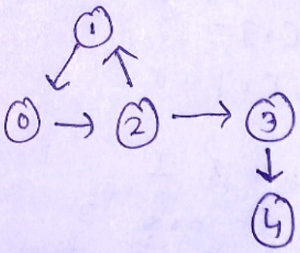
2 3 ⇒ SCC_2

but whole graph is not SCC as we start from

1 → 2 → 3

this is not possible in above graph so this is not SCC

Our algo



if we do DFS from last node we can find SCC.

DFS(4) ⇒ SCC_1 as we can't go anywhere from here

DFS(3) ⇒ SCC_2

DFS(2) → DFS(1) → DFS(0) ⇒ SCC_3

Let start from 0.

DFS(0)
↓
DFS(1) → DFS(4)
↓ ↑
DFS(2)
↓ ↑
DFS(3)

we did not get SCC we just traversed the graph.

algo steps

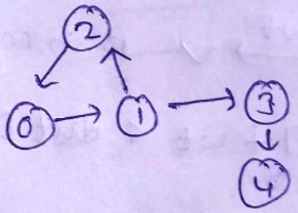
1) Sort all nodes based on their finishing time we studied in Topological Sort

2) Transpose graph

Do 1 → 4 as 4 → 1, this is transpose.

3) Apply DFS to ordering after Topo sort and Find / Count the result

DRY RUN



0
1
2
3
4

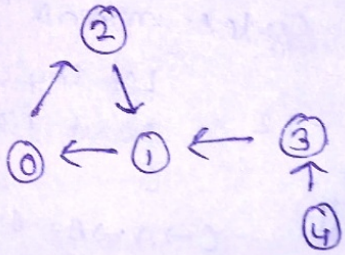
1) Do Topo Sort on graph make a stack, main()

DFS(0)
↓ ↑
DFS(1) ⇄ DFS(2)
↓ ↑
DFS(3)
↓ ↑
DFS(4)

return and push in stack if no where to go from here

2) Topo sort result is 0, 1, 2, 3, 4

Do Transpose of graph



3) Apply DFS on ordering

DFS(0)
↓
DFS(2)
↓
DFS(1)

now we can go back to 0 but its already visited so

SCC1 = {0 2 1}

and return

0, 1, 2 already visited so skip them from ordering

DFS(3)
↓
DFS(1) but 1 is already visited so

SCC2 = {3}

we go to 4

DFS(4)
↓
DFS(3) is already visited

SCC3 = {4}

So we have found our SCC1, SCC2, SCC3.

— x — x — x — x —

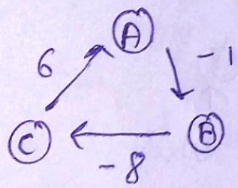
Lecture 101

Bellman Ford algorithm

It is used to find shortest path b/w source to distination but graph may contain negative weights.

Dijkstra's algo also finds shortest path but it does not work in negative weights. because it is a kind of Greedy approach.

Bellman ford does not work for -ve cycles. means



Let say we start from C

C → A → B = 6 + (-12) = -6

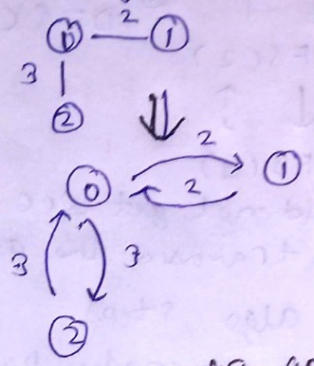
B → C = -6 - 8 = -14

C → A = -14 + 6 = -8

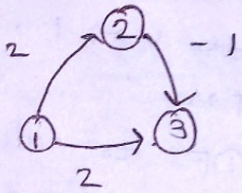
A → B = -8 - 12 = -20

everytime we get new shortest path and we are stuck in a cycle, this is called -ve cycle but using Bellman ford algo we can find whether -ve cycle present in graph or not

We can apply Bellman ford in Directed graph. but we can convert undirected Directed graph to directed



this way we can apply Bellman ford in undirected graph. Algo



1) Algo says we need to apply below formula (n-1) times in all edges.

$u \xrightarrow{wt} v$ is an edge

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

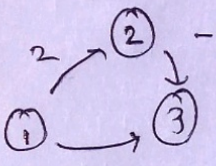
We apply it (n-1) times

1 → 2 (wt = 2)
2 → 3 (wt = -1)
1 → 3 (wt = 2)

(2) One more time apply same formula and if any distance gets updated means -ve cycle is present, we can't find shortest path

(3) Otherwise return the shortest distance

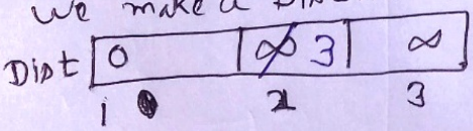
DRY RUN



edges
1 → 2 (2)
2 → 3 (-1)
1 → 3 (2)

n = 3 so we apply formula 2 times

we make a Distance array



Src node = 1
So dist[1] = 0, rest all ∞

1st time formula
we can start applying from any edge

a) dist[1] + 2 < dist[2]
0 + 2 < ∞ || True
{ Dist[2] = 3 }

b) dist[2] + (-1) < dist[3]
3 - 1 < ∞ || True
{ dist[3] = 2 }

c) dist[1] + 2 < dist[3]
0 + 2 < 2 || False
{ || Do nothing }

First time application of formula is done
— x — x — x — x —

Second time application of formula

a) dist[1] + 2 < dist[2]
0 + 2 < 3 || True
{ dist[2] = 3 }

b) dist[2] + (-1) < dist[3]
2 - 1 < 3 || True
{ dist[3] = 1 }

c) dist[1] + 2 < dist[3]

0 + 2 < 1 // false

{

// Do nothing

}

After 2 implementation

Dist array is

{ 0 , 2 , 1 }

1 2 3

↑ Src = 1

considering

1 → 1 = short dist = 0

1 → 2 = shortest dist = 2

1 → 3 = " = 1

2) checking negative cycle

apply formula again

for all edges

a) dist[1] + wt < dist[2]

0 + 2 < 2 // false

b) dist[2] + (-1) < dist[3]

2 + (-1) < 1

1 < 1 // false

c) dist[1] + 2 < dist[3]

0 + 2 < 1

2 < 1 // false

No, -ve cycle present

3) return dist[1]