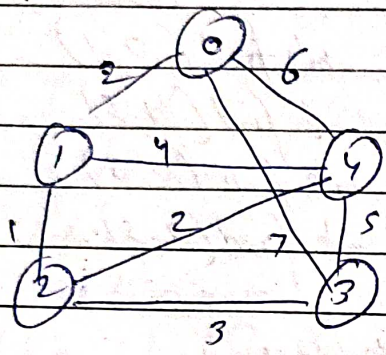
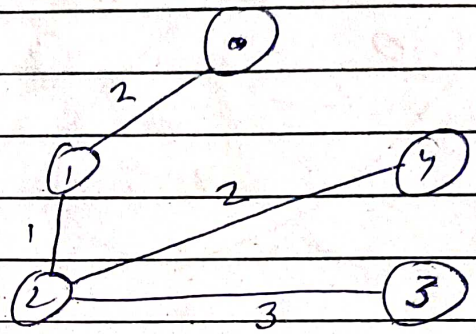
DATE : / 20
PAGE No

Prims Algo

- MST - Shortest path/weight to connect all vertices (V)
  - edges must be V-1





- There can be many Minimum spanning tree for a graph.

Approach

- There are many ways to implement prim's algo
- We have to find the minimum weight to connect all vertices in a graph.

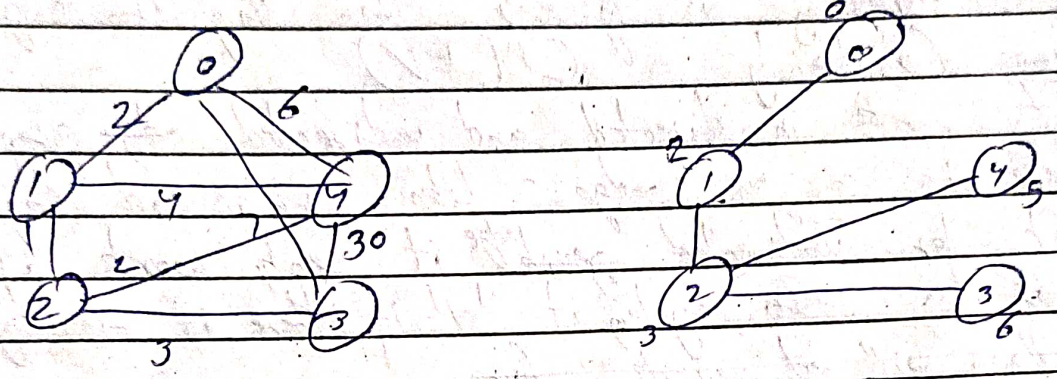
We put elements (node, weight) in priorityqueue and they give us the node that is smallest weight joint to our current node.
If that node is visited then we do nothing else we add the weight in our answer and do same process again until visited array not becomes the size of vertex.

- Greedy approach
- Time complexity = O(E log V)

DATE : / 20
PAGE No

Dijkstra Algo

They give us the smallest path from the given node to all the other nodes.



- Shortest path -

0	1	2	3	4
0	2	3	6	5

Approach

- Fill array with maximum value bcz we have to find minimum. Source index is 0 bcz we started there
- Pop element Push start vertex in priority queue.
- Pop element(node, weight), traverse ~~with~~ its neighbours, temp = add neighbour weight, ~~after adding if to the arr we are~~ + pop element weight (this is the weight that need to reach neighbour node) if this temp weight is smaller then the weight (that filled in array) then change old weight with new weight (temp ⇒ smaller weight) and add this element/node into pqueue.
- Similar to the prims algo just maintain the array that keep track of shortest weight.
- Time Complexity = O(ElogV)

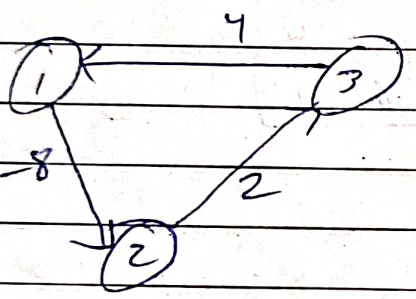
Optimization - We don't need pair bcz we have to traverse every edge

DATE : / 20
PAGE No

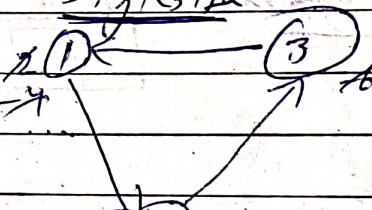
Bellman - Ford Algo

- Similar to dijkstra Algo
- Dijkstra algo doesn't work for -ve edges.
- Time Complexity is greater than dijkstra algo.
- Time Complexity = $O(EV)$
- It work in directed and undirected edge graph if there is a +ve edge ; like prim's and dijkstra.
  But for -ve edge it only work if there is a +ve cycle it tell if there is -ve cycle.

~~Bellman will not work for~~



Dijkstra



- If we traverse 1 time, one relax we surely found.
- For St. arr[5]=0

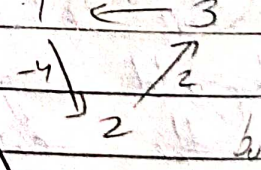
TLE & rem. n-1 we have to traverse

$1 \to 2$
$2 \to 3$
$3 \to 1$

4+2-8 => -2

This is effective edge wgt is -ve

So there is -ve cycle.



eff. edge
No -ve cycle
but -ve edge

Floyd Warshall also detect -ve cycle.

count/index	1	2	3
1	0	-2	-8	-6
2	-4	-10	-8
3	-6	-12	-10

(value decrease)
(-ve cycle test)

- Traverse the edges V-1 times (for worst ordering of edges)
- Traverse one more time, if any value decrease means there is a -ve cycle.
- Shortest path from source to all its vertices. (Default fill max)

DATE : / 20
PAGE No

Kruskal Algo.

- We have to find minimum spanning tree like prim's algo
- In this approach we use union find operation.
- union take 2 input and find their representative if their representative is equal means they lies in same set which means their is a edge between these 2 vertices. we have array for this.

index	0	1	2	3	4	5
arr	0	1	2	3	4	5

- Initially they are their own representative

union (2, 4)

index	0	1	2	3	4	5
arr	0	1	4	3	4	5

arr[2] = 4

4 is representative of 2

union (1, 5)

index	0	1	2	3	4	5
arr	0	5	4	3	4	5

arr[1] = 5

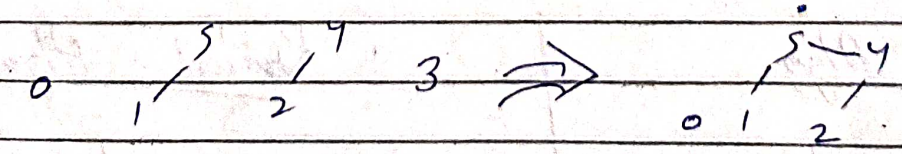
union (2, 5)

arr	0	5	4	3	5	5

arr[2] = 5

arr[4]

4

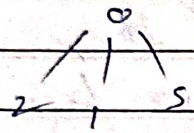


- If he is his own representative then y become parent of x.

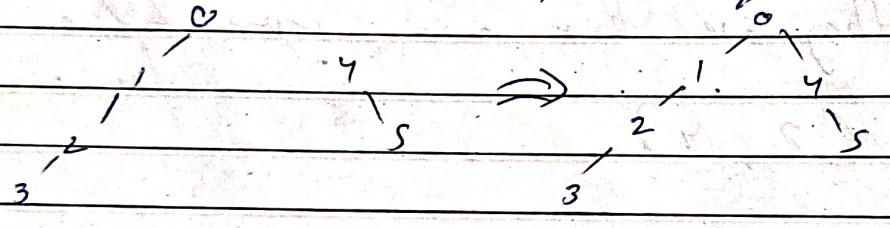
DATE : / 20
PAGE No

As we said y become parent / representative of x every time, it can increase time complexity. We use this two methods to optimize.

- Path Compression – every value contain representative of their set.



- Union by rank – Whose rank (height of x) between x and y ↑ is greater become the parent of another variable. If rank equal, then don't matter, whose ever become parent, parent rank ++.



Approach

Time Complexity = $O(E \log E)$

- Sort edges by their weight.
- If both vertices (a — b) has same representative means they have traversed and visited, if not then add it into set means join them means their representative become same. Add this edge weight into answer.
- Traverse until edges visited ~~vertices~~ not equal to V − 1.

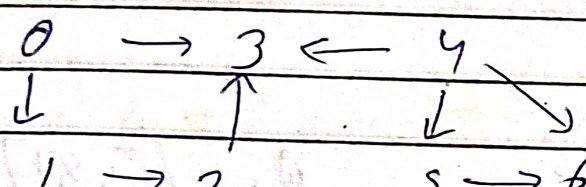
#	Prims Algo	Kruskal Algo
	• Generate MST from root vertex.	• Generate MST from least weighted edge
	• Select the shortest edge connected to root vertex.	• Select the next shortest edge.
	• $O(E \log V)$	• $O(E \log E)$

DATE : / 20
PAGE No

Kahn's Algo

Topological Sort using BFS.
we already covered Topological sort using DFS.

- There can many many solution after performing topological sort



index	0	1	2	3	4	5	6
inorder	0	1	1	3	0	1	2

- 0 inorder means no incoming edge on this vertex, we ~~can print this and~~ added into queue for future use.
- Queue contain vertex that has no incoming edge or inorder is 0, print this vertex and imagine like we have removed this vertex from graph so it's neighbour's inorder -- bz no outgoing edge from current to neighbours vertices.
- Do step two until queue becomes empty.

(print)	index	0	1	2	3	4	5	6	(queue)
0, 4	inorder	0	1	1	3	0	1	2	(0, 4)
1, 5	—	0	0	1	1	0	0	1	(1, 5)
2, 6	—	0	0	0	1	0	0	0	(2, 6)
3	—	0	0	0	0	0	0	0	(3)

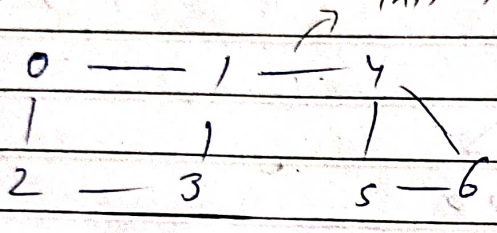
Ans = 0, 4, 1, 5, 2, 6, 3

DATE : / 20
PAGE No

Critical Edges / Bridges / Cut Edge

Means if a edge is removed from graph then graph is not connected.

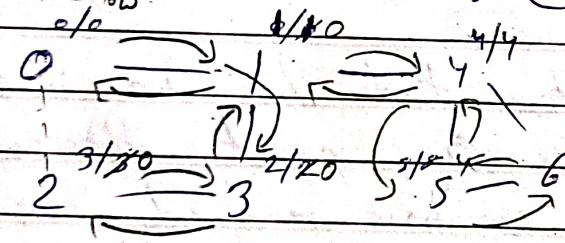
We have to calculate no of critical edges in a graph



Ans = [[1, 4]]

Maintain two array = low[] and time[]

- time increases whenever we go to any node
- low maintain the low value that if any node has a backward edge (then it become low) to its ancestor/parent.



TC = O(V+E)

SC = O(V)

- If neighbour is parent then just skip
- Fill max value default value is low.

While going to neighbour

- If parent then skip
- If visited then fill low value
- If unvisited then traverse and fill low value

while backtrack

If low[nbr] > time[node] then include in ans

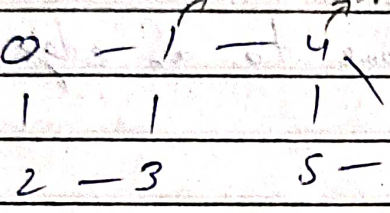
- Initially low[node] = time[node] = timer; if back edge then low[node] becomes "low".

DATE : / 20
PAGE No

Cut Vertex / Articulation point

If we remove an vertex the graph will break down into 2 components.

Calculate the articulation point



If we remove 1 then graph breaks into 2 components, as same as 4.

Ans = [1, 4]

- Traverse for unvisited vertex (graph can be not connected)
- Do dfs (vertex, parent)
- for root the parent is imaginary (-1).
- Main a set / boolean array bcz there can be cases that a vertex is articulation point for different components.
- Take a timer ~~re~~increment every time a unvisited vertex occur
- While traversing neighbour in dfs
  - if parent then skip
  - if visited just fill the low value = min(low(this), time(nbr))
  - if unvisited then traverse, now we know future(child) low value so fill in this also
- We maintained low and time array
- low array indicates how low (fast) can we reach, time indicates at what discovery time this vertex discovered.
- If low[nbr] >= time[node] && parent != -1
  in ideal case it has to be, if there is no back edge.
  We don't take the root bcz it can't decide ~~into~~ graph in upper half or lower half
- If child > 0 && parent != -1, imagine tree, root essential

DATE : / 20
PAGE No

Eulerian path / circuit

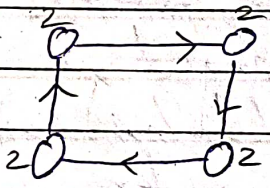
Eulerian Path and Circuit

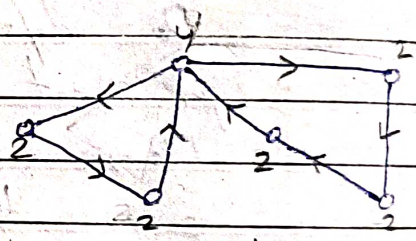
- Eulerian Path

- We have to traverse all edges
- From one node to another node we have to traverse but we can't come to an edge if it is visited.

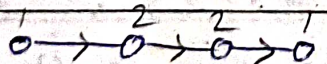
Approach

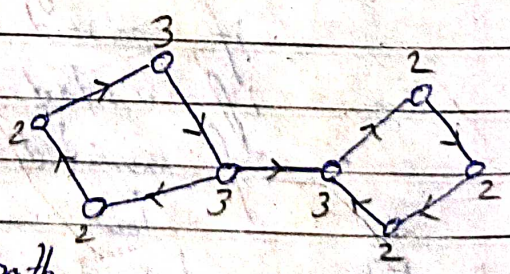
- We have to come to an vertex from another vertex via edge and we have to go to other vertex from current vertex.





- So there must be even edges from n vertex
- Or there must be two 2 odd edges. Bcz if we go from an vertex we should be end on another vertex.





- We don't have to care about path.

- Eulerian cycle

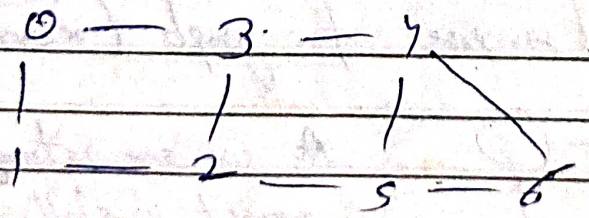
- Where we start from, must end there. (n even edges)
- Eulerian cycle is eulerian path also.

DATE: / 20
PAGE No

Hamiltonian path and cycle

- From vertex can we reach all the other vertices in a graph, if we reach vertex only one time.
- Traverse DFS if neighbour is not visited.
- Maintain visited array or set.
- We have to check for every vertex means we have to start from all the verteces, one at a time. Give a chance to a node/vertex to be the starting vertex.
- Also after traversing all the neighbours we have to remove this vertex from visited set bcz these can be any other possibility also that this vertex be the true at any other point.
- If all vertices visited then this is a ~~ham~~ hamiltonian path.

- For hamiltonian cycle the last vertex ~~be~~ should be the neighbour of the start vertex.



Hamiltonian path
{0,1,2,3,4,5,6}, {0,1,2,3,4,6,5}, {0,1,2,5,6,4,3}, {0,3,4,6,5,2,1},

Hamiltonian cycle
{0,1,2,5,6,4,3}, {0,3,4,6,5,2,1}

- Hamiltonian cycle is also hamiltonian path.

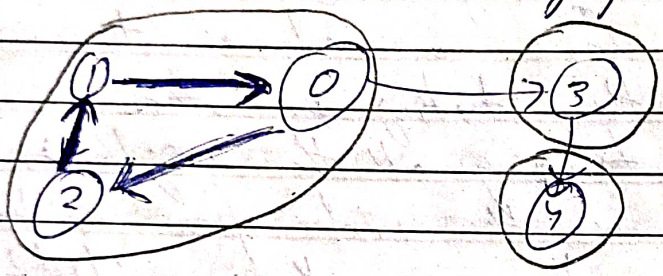
DATE : / / 20
PAGE No

SCC Kosaraju

Kosaraju Algorithm

- Strongly connected component

- A directed graph is strongly connected if there is a path between all pairs of vertices.
- A strongly connected component of a directed graph is a maximal strongly connected subgraph.



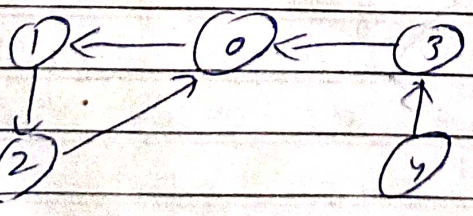
Ans = 3

- Approach

Step 1: Do DFS and fn topological sort store it into stack

0
3
4
2
1

Step 2: Make Transpose of graph (reverse edges)



As we see strongly connected component is still strongly connected component.

Step 3: Pop elements from stack and do dfs and mark true.

If will find its strongly connected component.
If pop element is visited means it already been covered in SCC.

DATE: / 20
PAGE No

Tarjans Algo

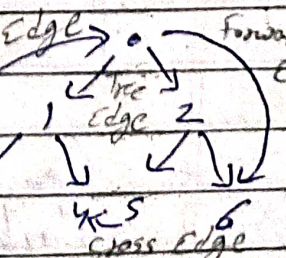
Kosaraju algo can't tell what are those SCC's.

Approach

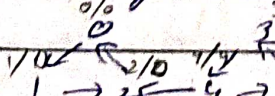
Step 1: Take the discovery time array and low time array.
Also take stack ~~for~~ which contains what vertices we traversed. and boolen array which contain what vertices present in a stack.
Fill disc time with -1 which tells whether we traversed vertex or not. So we didn't need another boolean array.

Step 2: Do dfs for unisited vertex.
Fill the ++ discovery time.
Fill (default) low time and
add vertex in stack and mark true, increase time.

Step 3: Visit neighbour
if visited then there are two case whether edge is a back edge or cross edge.
We take the low value from disc[neighbour]; Back Edge.



Cross Edge: We ignore it bcz this



SCC1 ← SCC2
low[4] become 2, SCC2 low become 2 (if taken) which is a part of SCC 1; which is not true.
Step 4: unisited: Traverse and take low from neighbour

Floyd Warshall

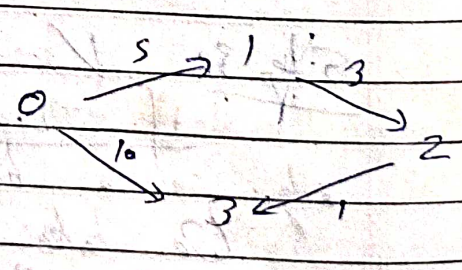
Step 4: If After traversing if low[source] and disc[source] is equal then this is the head of SCC. Pop elem
Pop elements from stack and check if it is equal to source, if yes then stop, else add into list.

- Step 3 disc[nbr] is the discovered vertex time low[nbr] not bcz there can be other child, which updated the low value.

Floyd Warshall -

- Used for solving the all pairs shortest path problem.
- Used to ~~path~~ find shortest path between all pair of vertices.

0	5	-1	10
-1	0	3	-1
-1	-1	0	1
-1	-1	-1	0



- Can detect -ve cycle also. A vertex to itself shortest path is always 0.

DATE : / 20
PAGE No

As we saw there's a path between 0→2 but there is no direct path.

- Approach

+ Traverse V time in a graph. ($V \times V$)
+ If index is k. (transit) means take that path from i to j.
  if i == k or j == k skip
  bcz we are taking that index
  or there is no path from i → k or k → j
  then also skip. bcz there is no effective path

e.g. i = 1 j = 2 k = 0 1 → 0 → 2 (no path)
i = 1 j = 3 k = 0 1 → 0 → 3
i = 0 j = 2 k = 1 0 → 1 → 2

- now we have take the path via k.
As we saw there's path -1 so we have to take max and then take minimum from (i → k → j, itself).

$$T.C. = O(V^3)$$

Dijkstra Algo	Bellman ford algo
• Greedy	• DP
• $O(E \log V)$	• $O(VE)$
• Don't work for -ve edge	• Work for -ve edge.
• Whole info. about vertices	• Contain info. about other vertices.

+ Bellman Ford & Floyd Warshall helps to detect -ve cycle also.