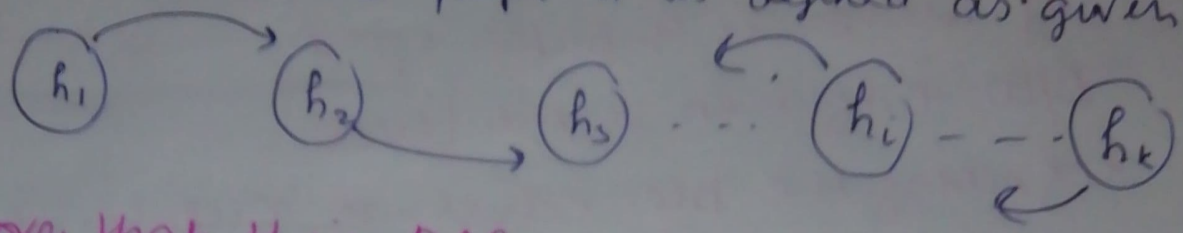
Q1

Let G be a Directed Graph such that $C_1, C_2 \dots C_k$ are strongly connected components.

Now construct the graph H is defined as given

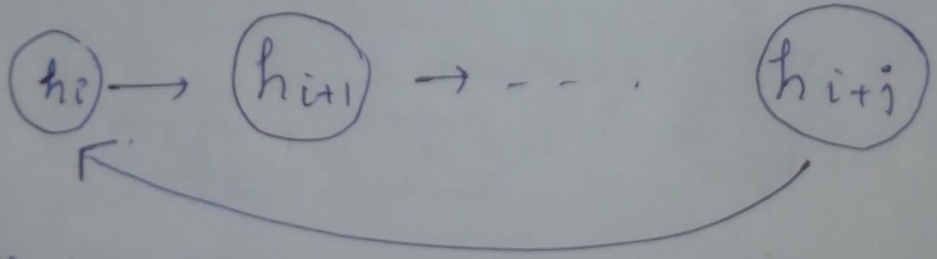


a) Prove that H is DAG

Proof by contradict^n

Let's assume that H is not a DAG if it is the case then there should be atleast 1 cycle that prevents H to be a DAG.

Let there are j nodes in cycle with lowest indices as i, then we could show cycle as



Now we can say that on original graph G, any two vertices u & v such that $\exists\, u \in C_k$ & $\exists\, v \in C_L$ where $i \leq k$, $L \leq i+j$ and $L < k$.

$\Rightarrow$ There is a path from u to v and at the same time v to u (as cycle provide 2-way connectivity).

$\Rightarrow$ if $\exists\, v \in C_L$ which has path to another vertex $u \in C_k$ then we know $\forall$ v in C_L has path to $\forall$ v in C_k (as if $u \to w$ & $w \to v$ then $u \to v$)

similarly we can say that each vertex from C_k to all vertices in C_i.

$\Rightarrow$ C_i & C_k should be in one SCC but as per our assumption this is not true

Hence we can say our assumption is wrong & H is a DAG.

b) Algo Design

By kosharaju's Algo find all SCC & at the same time create an array scc_info of size |V| in which the serial number of scc component is present.

Set a int variable scc_number to 0 & at last increment after each DFS call (which find SCC) & put the value of scc_info(v). = scc_number.

Algo 1

input : scc_info : int array of size |V|.
: adj list rep of G.
: H = array of set of size K.

procedure

Call kosaraju on G as specified above.

Step 1 — for each $u \in G$:

Step 2 —
  for all neighbour v of u in G:
    if scc_info[u] != scc_info[v]
    H[scc_info[u]].insert(scc_info[v]).
  end for

end for

Complexity Analysis:

for step 1 $O(V+E)$. traversal of G.

for step 2 $O(V+E)$. traversal of G

total $O(V+E)$

Proof of Correctness

- SCC_info array is designs as each vertex got value of component number
- After that is Algo 1 we iterate over each vertex on graph G & for each edge $u \to v$ if component number of u & v are different then it means there is an edge in ~~st~~ hi to hj st $u \in C_i$ & $v \in C_j$
- In above mensioned way it got all edges in H. & since we use SET datastructure to collect edges no duplicate edge will be present.

Q2.

Solution.

Similar to Q1, reduce the Graph G to SCC's structure to H

As we know these vertices in components are strongly connected to each other hence they are weekly connected too.

for vertices to be weakly connected across diff SCC H should be a PATH.

Algorithm & 2:

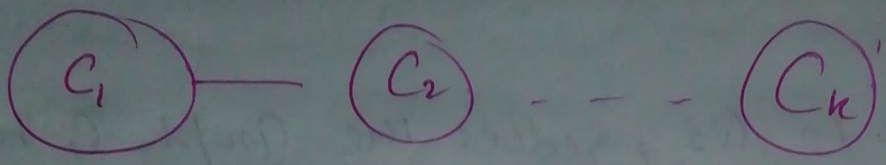
1. Call kosaraju's Algo on G as in Q1.
2. Call the Algo 1 from 1
3. Search the vertices v in H such that $\delta(v) = 0$ min
4. Start DFS(V) on H
5. check all vertices in H if they are visited.
   - if yes then G is weakly connected otherwise not.

Time complexity

E' V' are edges in H

step 1	$O(V+E)$
" 2	$O(V+E)$
" 3	$O(E') < O(E)$
" 4	$O(V'+E') < O(V+E)$.
" 5	$O(V') < O(V)$
total	$O(V+E)$

Proof of correctness.



$\forall$ v in C_i are strongly connected to $\forall$ v in C_i of H is a path then if atleast one path for each vertex to other in diffrent component too.

(3)

Intuition

need second shortest path for u to v.
apom routing table where each neighbour can tell the shortest path to another.

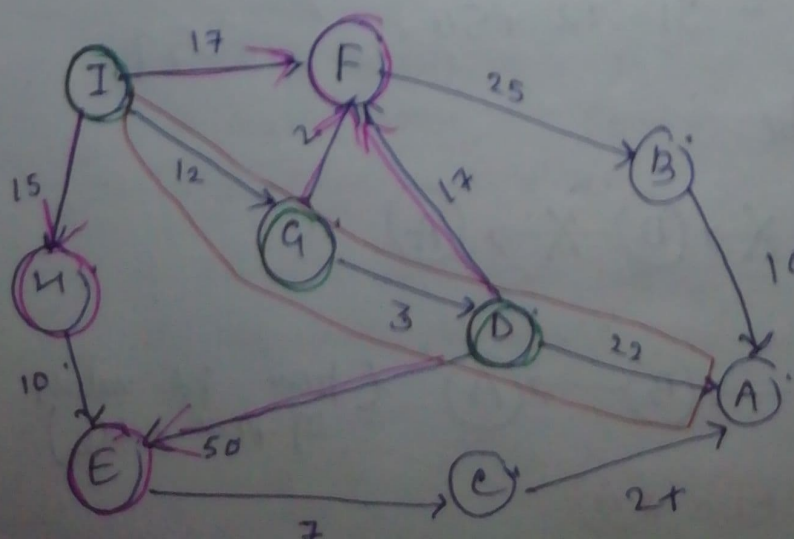
step 1 : 1st we need to know that with how much weight a can every node reach to sink vertex t.
(we can do it more clever way to reverse edge of graph (G^R) & doing dijkstra from v to u).

step 2 After that we iterate over the immidiate neighbours (n) of node (n_s) that belong to shortest path from v to u to compute Additional cost.

step 3 we select the node that offers least additional cost to have on second shortest path.

example for better understand let's take following example. where u = I & v = A. & we need to find shortest path from I to A.

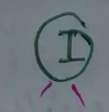
As per step 1 we ran dijkstra (A to I) on $G^{Rev.}$ & got highlighted path as shortest A - D - G - I

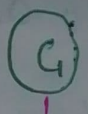


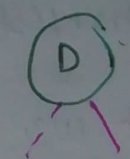
vertex	rev distance (rd)	prev vertex
A	0	-
B	10	A
C	24	A
D	22	A
E	31	C
F	35	B
G	25	D
H	41	E
I	37	F

- Shortest path in Red.
- nodes in shortest path are in green
- neighbours of nodes in shortest path are in pink

Additional cost = $h_d(n) - h_d(n_s) + w(n_s \to n)$

where n_s = 





n = F H F F E.

Additional cost calculation.

for n_s = I.

n = F cost = $h_d(F) - h_d(I) + w(I \to F)$.
$35 - 37 + 17 = 15$

n = H. cost = $41 - 37 + 15 = 19$.

for n_s = G

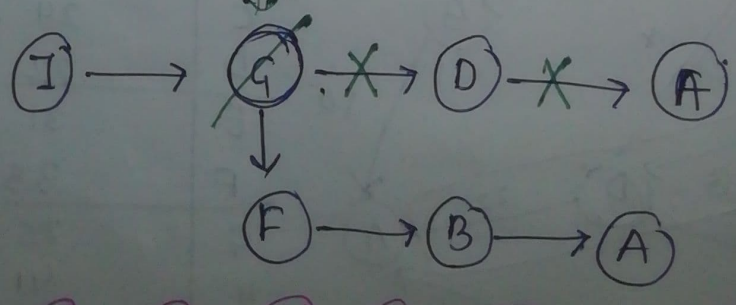
n = F cost = $35 - 25 + 12 = 22$.

for n_s = D.

n = F cost = $35 - 22 + 17 = 30$

n = E cost = $31 - 22 + 50 = 59$

Shortest path will be.



(from h_d values of prev table).

Ans I → G → F → B → A

Algorithm

Step 1: Do Dijkstra ($v \to u$) to G^R to get reverse shortest path

Step 2: for each node u' in shortest path

for each node v' adj of u' not in shortest path

O(1) ← calculate the cost as $rd(v') - rd(u') + w(u' \to v')$

end for

end for.

Step 3: Select the min of cost for node k adj to k_s.

Step 4: output the cost of 2nd shortest path as [cost of original shortest path + cost (k)].

Step 5: print out the path from (u to k_s) + ($k_s \to k$) + (k to v using the table of G^R)

Time complexity

Step 1	$O(V + E \log V)$
2	$O(V + E)$.
3	$O(1)$
4	$O(1)$
5	$O(V)$.
total =	$O(V + E \log V)$

Proof of correctness

value of additional cost will never be -ve as node n is not present in shortest path & $n_s + w(n \to n_s)$. always have more value

if additional cost is 0 then it means of a second shortest path which's weight is equal to shortest path

(4)

Algorithm

Step 1: Select the vertex v in DAG (D) with $\delta^-(v) = 0$

Step 2: Do BFS (v) on D

Step 3: for set of vertices in same level of BFS(v) in D. remove the tree edges in D and repeat step 1 and 2. if ~~if~~ edge E ~~of V~~ present in a level.

Step 4: print out the result of each BFS in step 3.

Time complexity

Step 1: $O(V+E)$

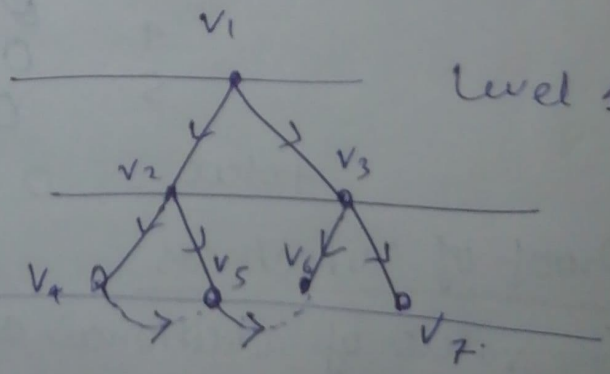
2: $O(V+E)$

3: $O(V+E)^2$

4: $O(1)$

total = $O(V+E)^2$

Example



Set 1 v_1

Set 2 v_2 v_3

v_4 v_5 v_6 v_7

Set 3 v_4 ~~v_7~~

Set 4 v_5

Set 5 v_6

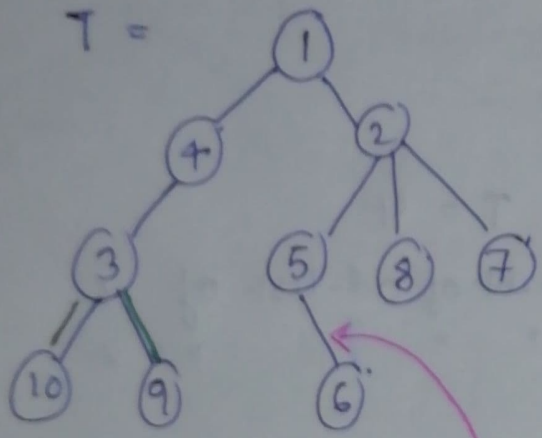
(5) Intuition

→ BFS uses queue to store intermidiate neighbours which follow FIFO

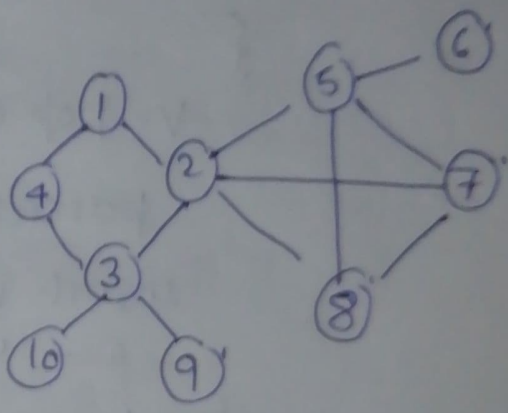
→ our idea is to start ~~graph~~ level wise traversal of tree & put vertices in a stack that follows LIFO, so we are buiding tree from the bottom using graph & adjacency list representation of graph is designed simultaneously.

example

T =

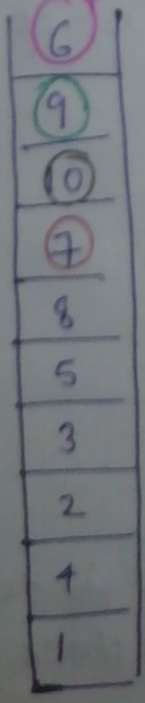


G=



Stack ↓ removed

got this edge



edges from G-T	edges formed using T/stack	3
	4, 2	1
3	1, 5, 8, 7	2
2	4, 10, 9	3
	1, 3	4
7, 8	2, 6	5
	5	6
8	2	7
7, 5	2	8
	3	9
	3	10

- next Do the BFS (root[T]) in G defined by prev. Ad-list.
- compare the parent array for Tree so obtained with previous step & given T.
  if Parent-{BFS (root(T)) on G} = parent-(T) then this ordering is possible otherwise not.

Algorithm.

Step 1: fill the stack using level ordering of T.
step 2: while (stack is not empty).
{
V = pop (stack).
parent-(V) in T = u.
insert v at the start of ad-list of u
" u " " " " " " v.
}
step 3: for each edge u → v in G-T edges.
insert v at the start of ad list of u
" u " " " " " " v
end for.

Complexity.

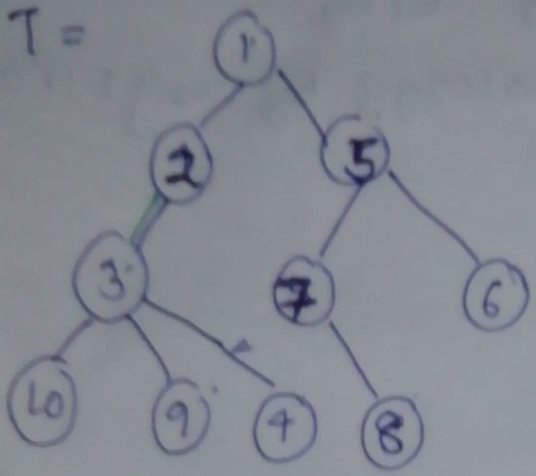
step 1	O(V)
2	O(V)
3	O(E)
total	O(V+E).

Proof of correctness. we are using what BFS is doing in reverse BFS is using queue & we are using stack to get back representation of G using T

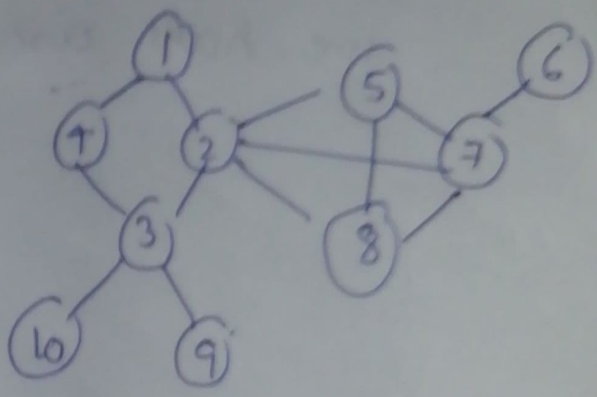
(6)

- replace the stack with queue in Qust^n 5.
- in step 2 & 3 insert at the end of Ad-list
- rest steps will be same that we will be showing using same graph by DFS.

T =



G =



Queue gd in level ordering of T.

1	2	3	5	10	9	4	7	6	8	

Adj list of G. designed as other edges.

	edges from a
1	2
2	1, 3, 5
3	2, 10, 9, 4
4	3
5	2, 7, 6
6	5.
7	5, 8
8	7
9	3
10	3

make DFS on this Ad-Mat & check the parent of it with T.'s parent array

if both of them are same then this ~~DFS~~ $_T$ can be get using G. by using this adj list

time complexity

same as nothing changes in terms of implementing & iterating structure $O(V+E)$.

Proof of correctness

DFS uses stack ⇒ LIFO

we have used reverse strategy by using FIFO