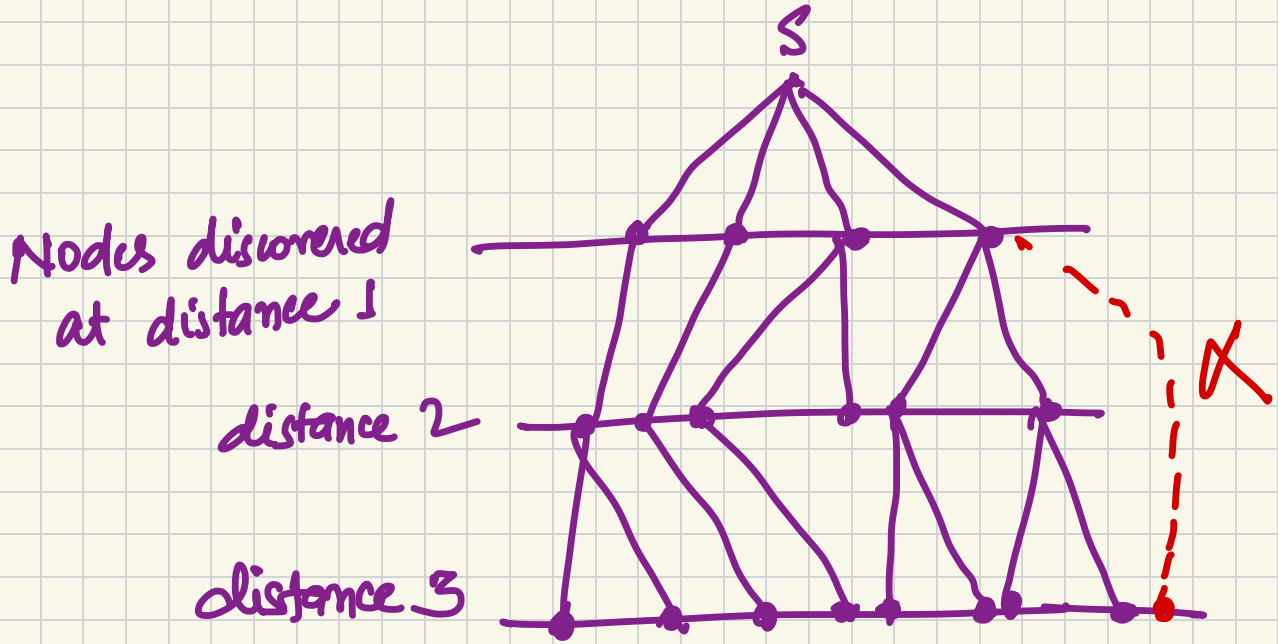
ACOL 202

lecture 27

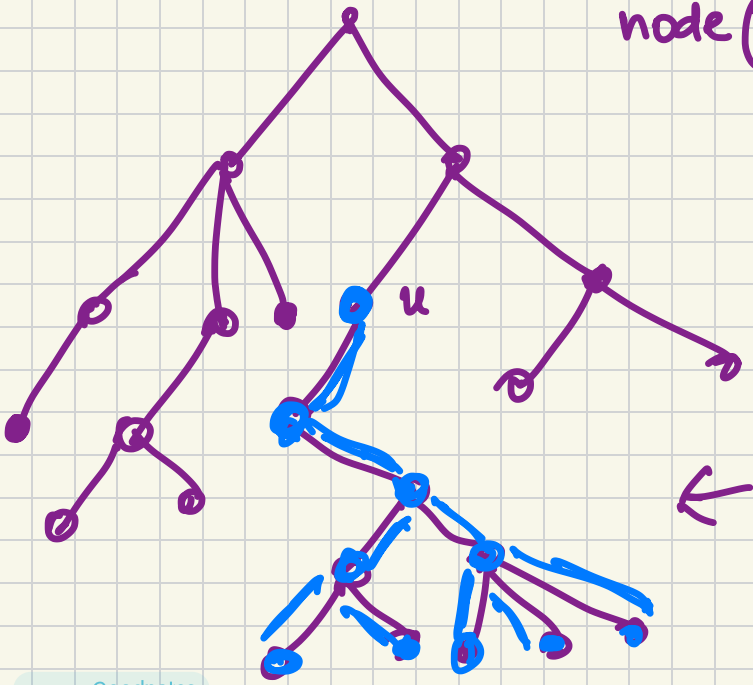
(6th May 2025)

Tree BFS Tree



Subtree

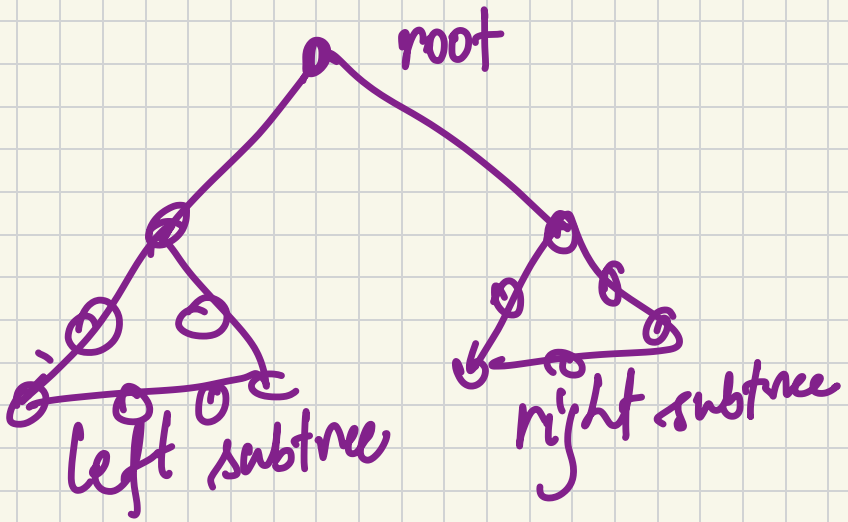
subtree rooted at a node is all the nodes that are below that node (including self, child, child of a child, child of a child of a child ... and so on — descendants) and the induced subgraph of those nodes.



← The subtree rooted at u is all the nodes and the edges shown in blue here.

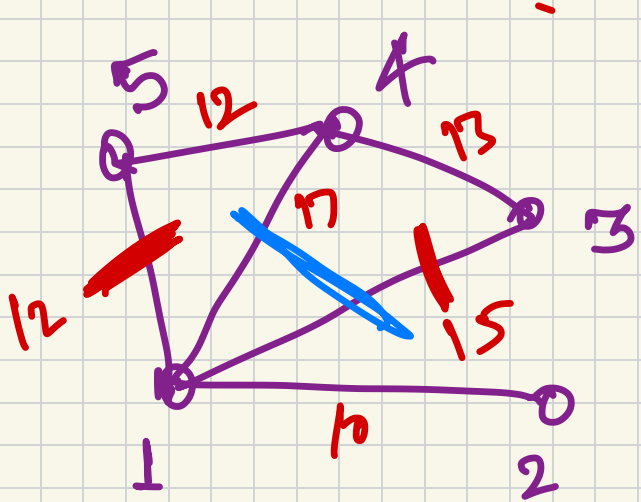
Tree traversals

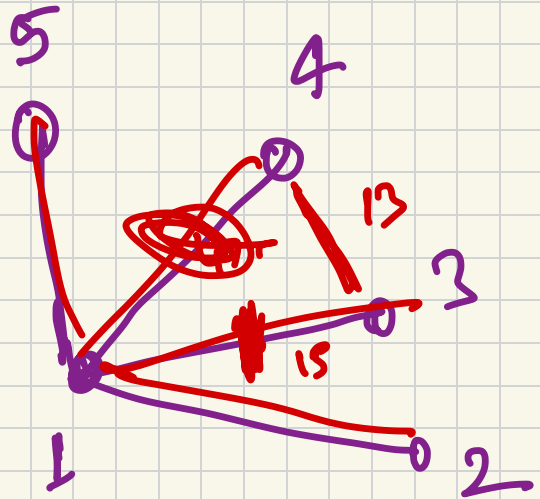
- pre order traversal
- inorder traversal
- postorder traversal



- root , left subtree , right subtree
- left subtree , root , right subtree
- left subtree , right subtree , root

Spanning Tree





Directed Graphs

Strongly connected

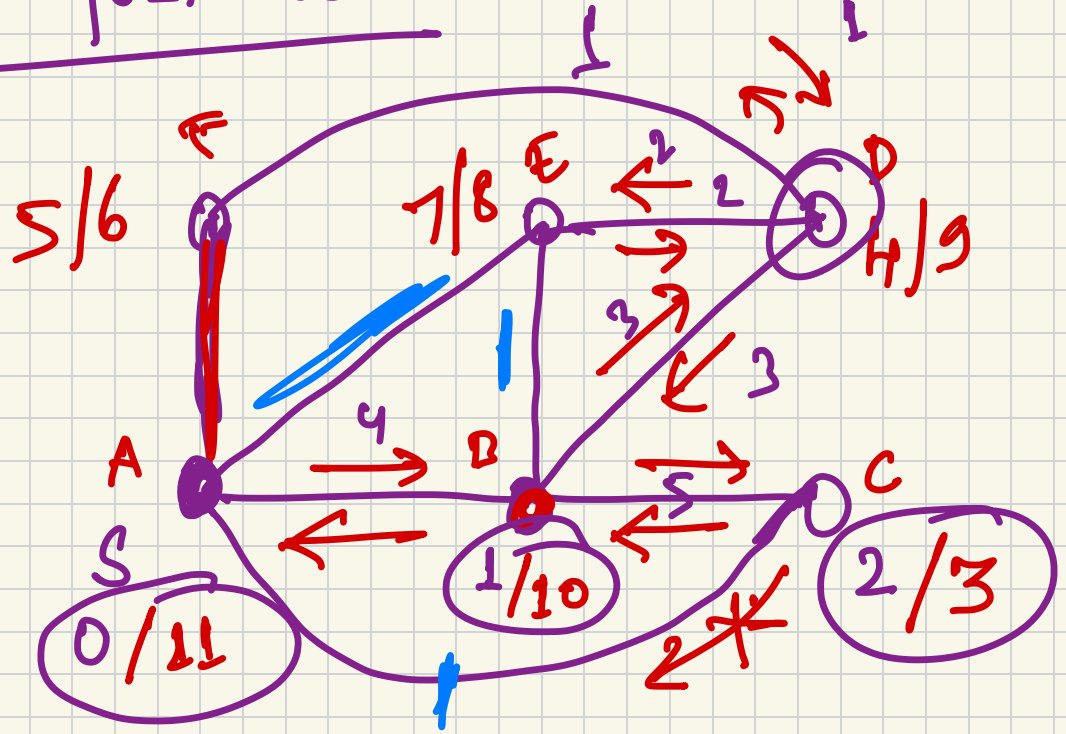
Strongly connected components

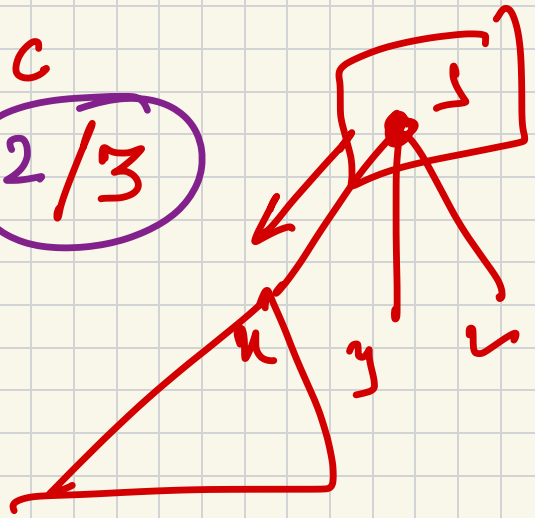
A SCC of $G = (V, E)$ is a set $C \subseteq V$ such that

i) for any $u, v \in C$ u and v should be strongly connected.

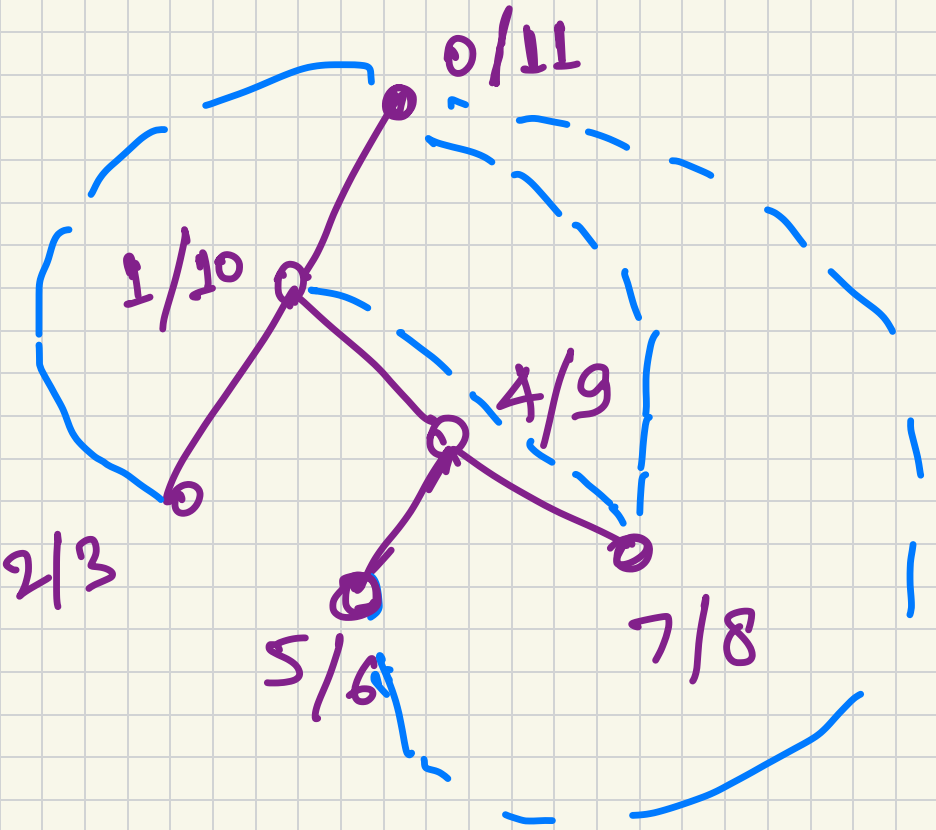
ii) for any $x \notin C$ $(x \in V)$, x should not be strongly connected to every $c \in C$.

Depth-first search





How can we implement this recursion!



Tree edges

Back edges

Dijkstra's algorithm (shortest path)

$d[s] = 0$

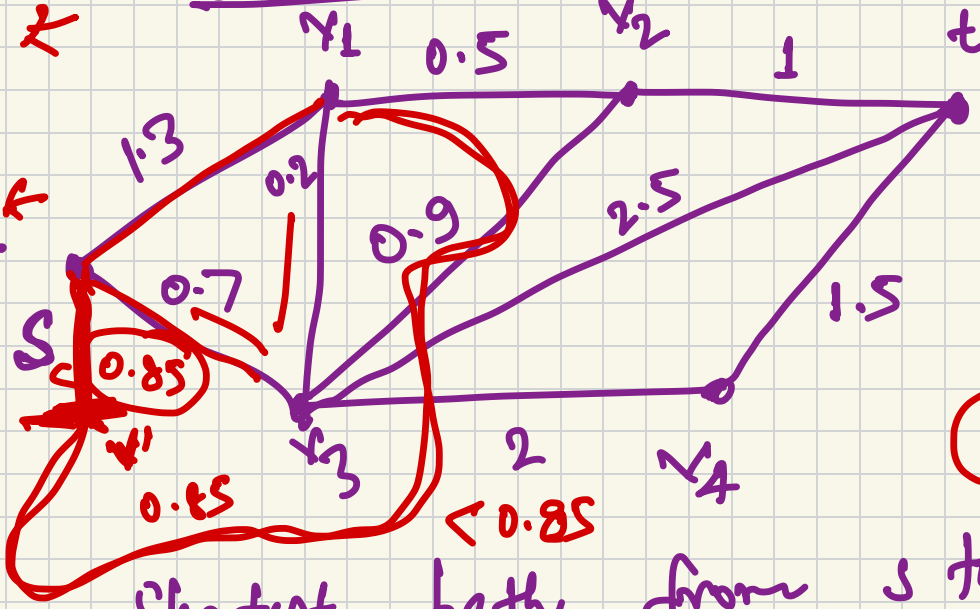
$d[v_1] = 1.3$ ←

$d[v_2] = \infty$

$d[v_3] = 0.7$ ←

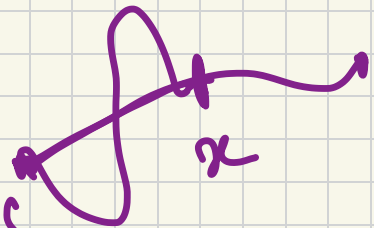
$d[v_4] = \infty$

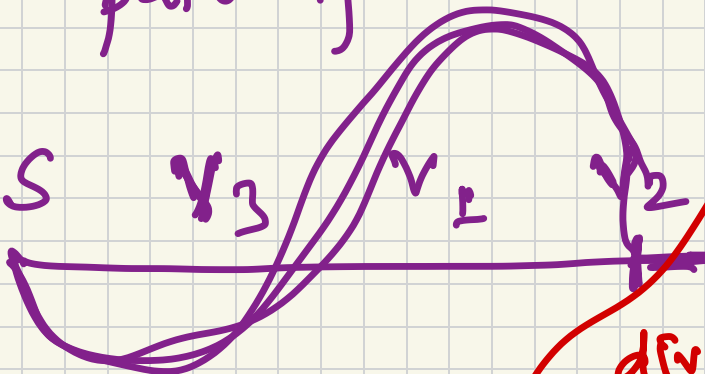
$d[t] = \infty$

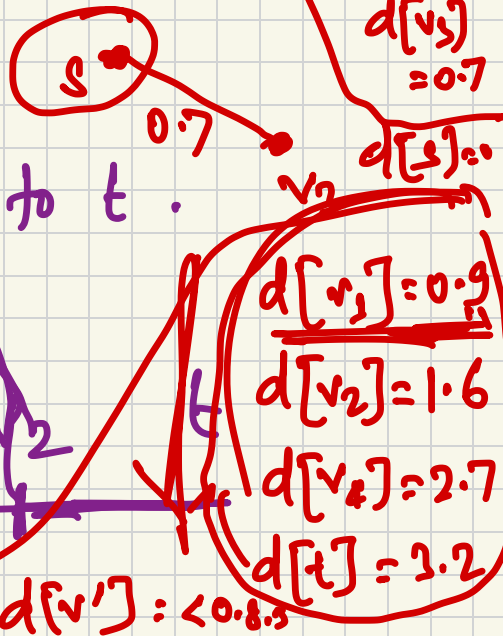


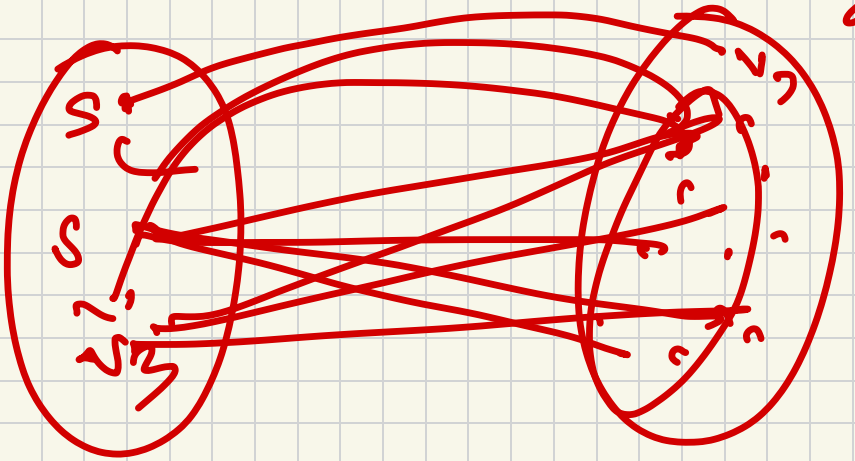
All the edge weights are positive

Shortest path from s to t.









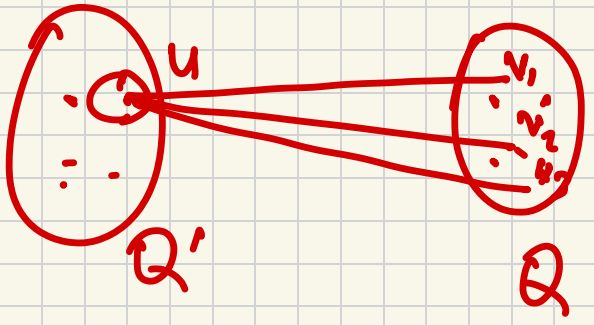
$d[v_7]$

v_3

edge relaxation

$d[v_1]$?

$d[u] + wt(u, v_1)$



Q

$d[v_1] \leftarrow d[u] + wt(u, v_1)$

How does the algorithm work?

Let s be the source vertex.

The idea is to compute the shortest distance of every vertex from s.

Let d[v] denote the shortest distance of a vertex v from s.

We wish to accurately compute the values of d.

The algorithm maintains an upper-bound on the shortest distance to every vertex (from s), and refines it iteratively.

The design of the algorithm relies on this property of shortest paths that if

the path shown above is a shortest path from s to t then for any vertex x on the path, the segment of the path from s to x is also a shortest path from s to x.

If there was a shorter path from s to x, we could have taken that path to reach x and then continued from x to t in the path shown above to get a shorter path to t.

Initially, the d value is 0 for s, and ∞ for every vertex other than s.

There are two sets of vertices — Q and Q' — that we maintain. Initially, Q contains all the vertices and Q' is empty.

In every iteration, the algorithm claims to have accurately found the shortest distance to one vertex in Q, and moves it to Q'.

→ Which is the vertex that is moved to Q'?

We move u, if d[u] is the minimum of all the d-values (for vertices) in Q.

Once the vertex is moved, the d values are updated to a tighter bound if possible.

→ which are the vertices whose d values may have to be updated if u is moved to Q'?

All the vertices that are adjacent to u and in Q.

→ How do the d values get updated?

For a vertex v that is adjacent to u, if the current d[v] is bigger than (d[u] + weight(u,v)) then d[v] is updated to (d[u] + weight(u,v)).

In other words, we have found a tighter upper bound for d[v] and therefore it is updated.