Algorithms and Data Structures

Graphs 3: Finding Connected Components

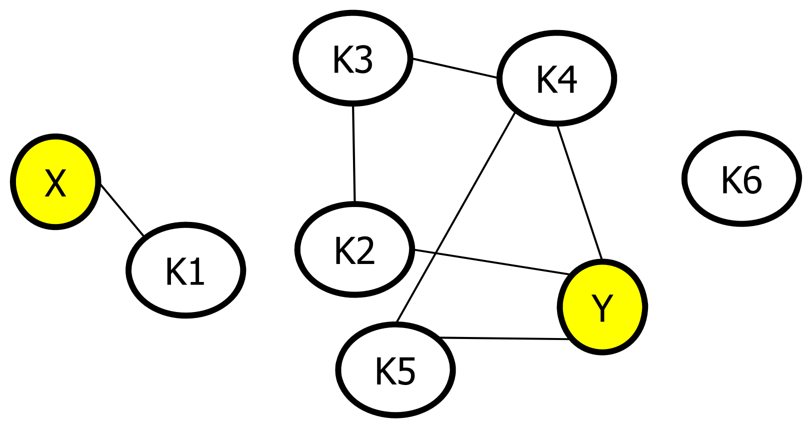
Marius Kloft

Content of this Lecture

- Finding Connected Components in Undirected Graphs
- Finding Strongly Connected Components in Directed Graphs
  - Why?
  - Pre/Postorder Traversal
  - Kosaraju's algorithm

Reachability

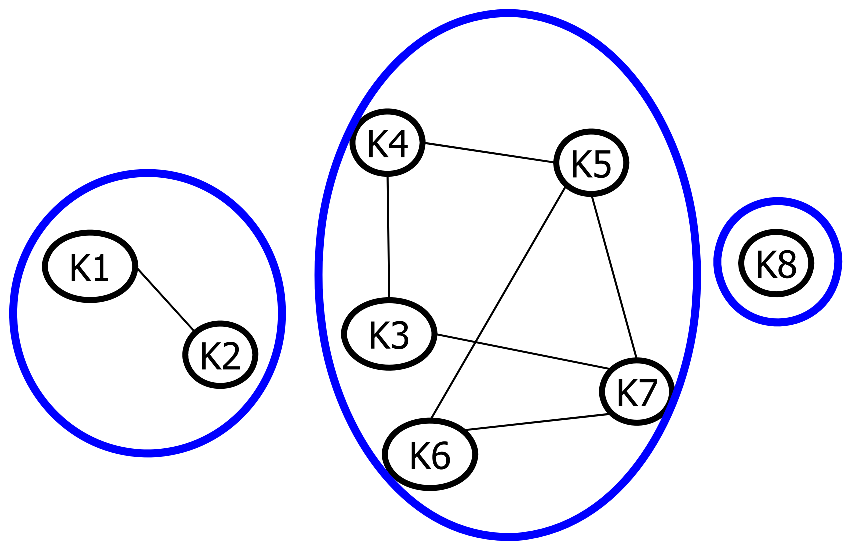
- Given a graph G, can we reach node Y from X?



- Solution for undirected graphs: Compute connected components of G

Recall Definition of Connected Components

- *Let G=(V, E) be a graph.*
  - *An induced subgraph G'=(V', E') of G is called connected if G' contains a path between any pair v,v'∈V'*
  - *G' is called maximally connected, if no subgraph induced by a superset of V' is connected*
  - *Any maximal connected subgraph of G is called a connected component of G*



Finding Connected Components in Undirected Graphs

- In an undirected graph, whenever there is a path from r to v and from v to v', then there is also a path from v' to r
  - Simply go the path r → v → v' backwards
- Thus, DFS (and BFS) traversal can be used to find all connected components of an undirected graph G
  - Whenever you call traverse(v), create a new component
  - All nodes visited during traverse(v) are added to this component
  - Complexity: O(n+m)

Called once for every connected component →

```
func void DFS ((V,E) graph) {
  U := V;          # Unseen
nodes
  S := ∅;          # Seen nodes
  while U≠∅ do
    v := any node from( U);
    traverse( v, S, U);
  end while;
}
```

In Directed Graphs ("Digraphs")

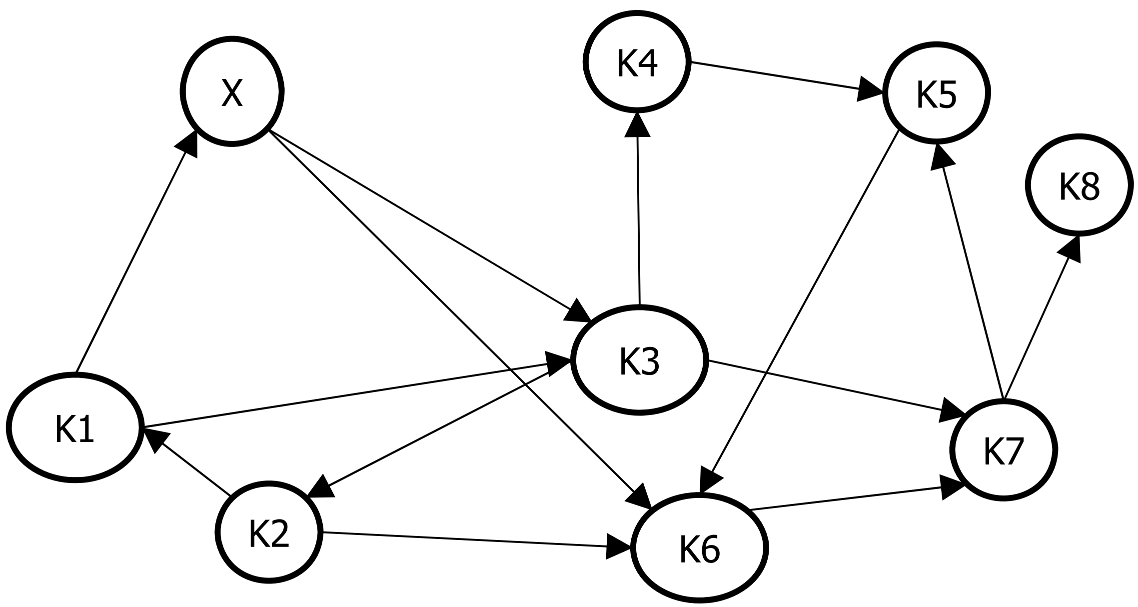
- The problem is considerably more complicated for digraphs
- Still, it will turn out:
  - Tarjan's or Kosaraju's algorithm find all strongly connected components in O(n + m)!

Content of this Lecture

- Finding Connected Components in Undirected Graphs
- Finding Strongly Connected Components in Directed Graphs
  - Why?
  - Pre/Postorder Traversal
  - Kosaraju's algorithm

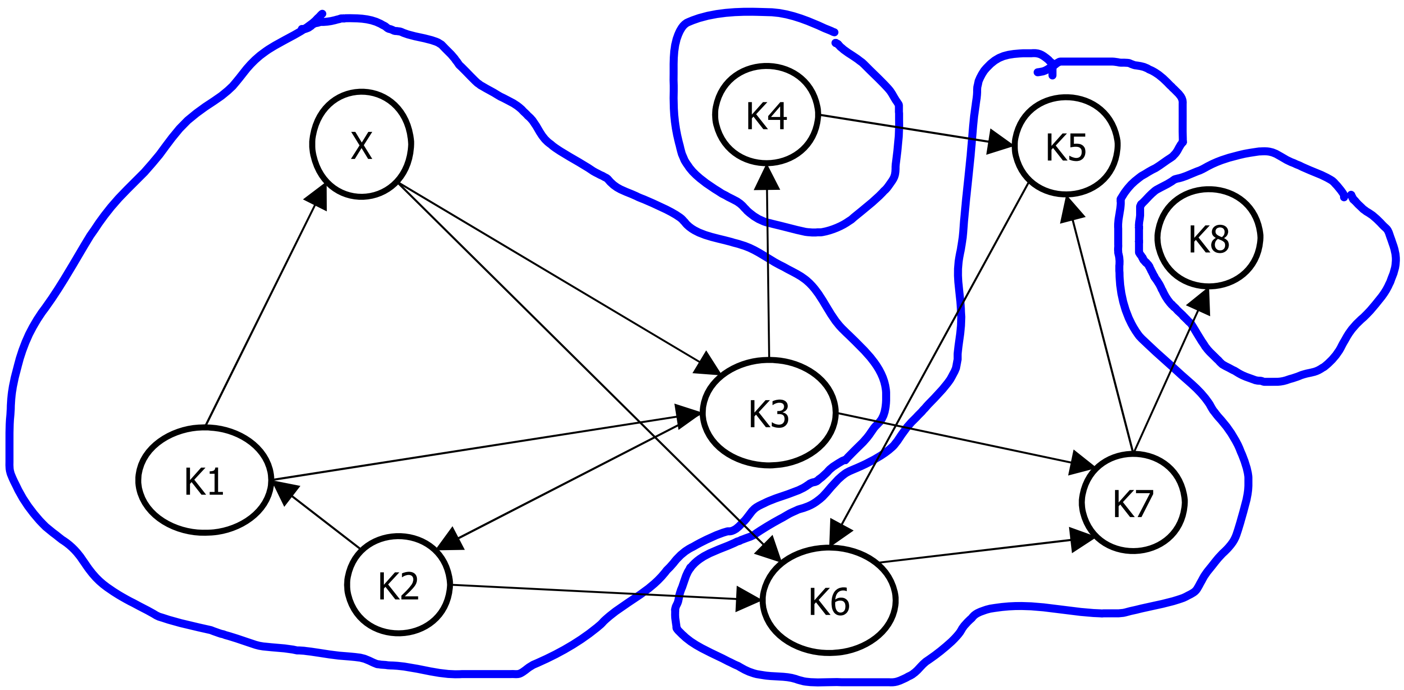
Recall Definition of Strongly Connected Components

- *Let us now be given a directed graph G=(V, E).*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



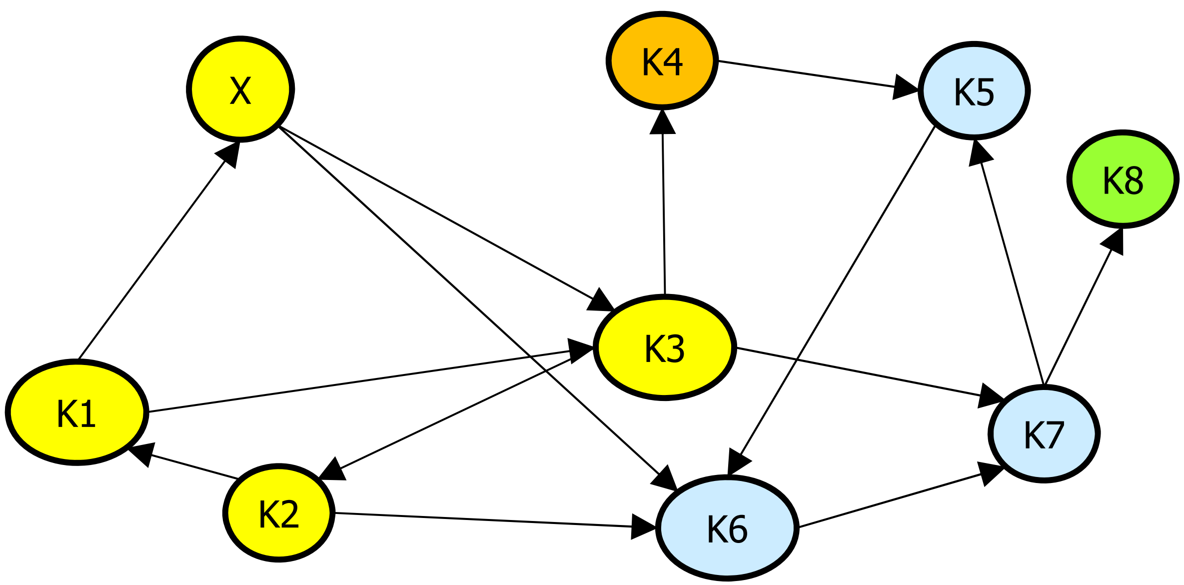
Recall Definition of Strongly Connected Components

- *Let us now be given a directed graph G=(V, E).*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



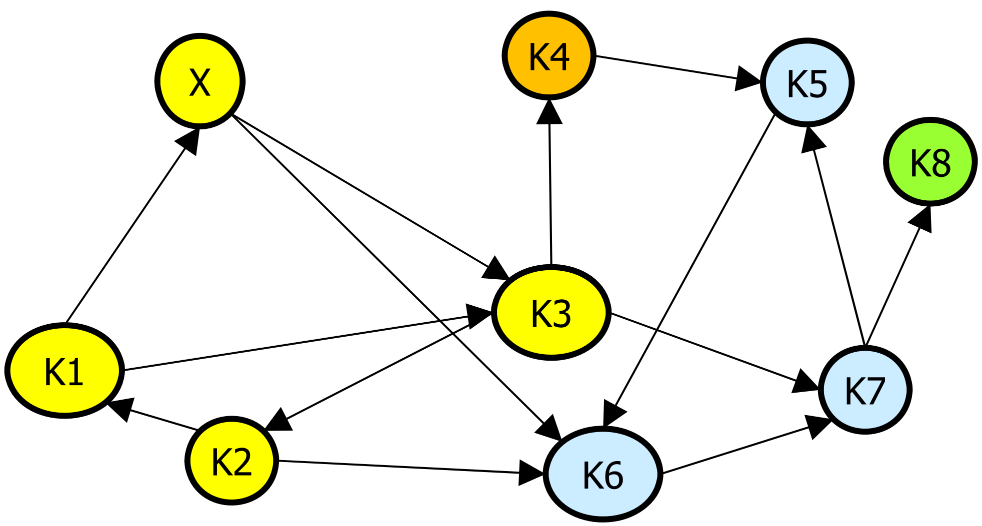
Recall Definition of Strongly Connected Components

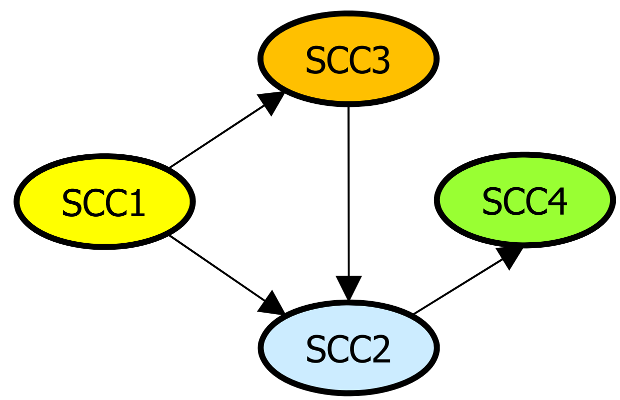
- *Let us now be given a directed graph G=(V, E).*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



Why? Contracting a Graph

- Consider finding the transitive closure (TC) of a digraph G
  - If we know all SCCs, parts of the TC can be computed immediately
  - Next, each SCC can be replaced by a single node, producing G'
  - G' must be acyclic – and is (much) smaller than G
  - Intuitively: TC(G) = TC(G') + SCC(G)



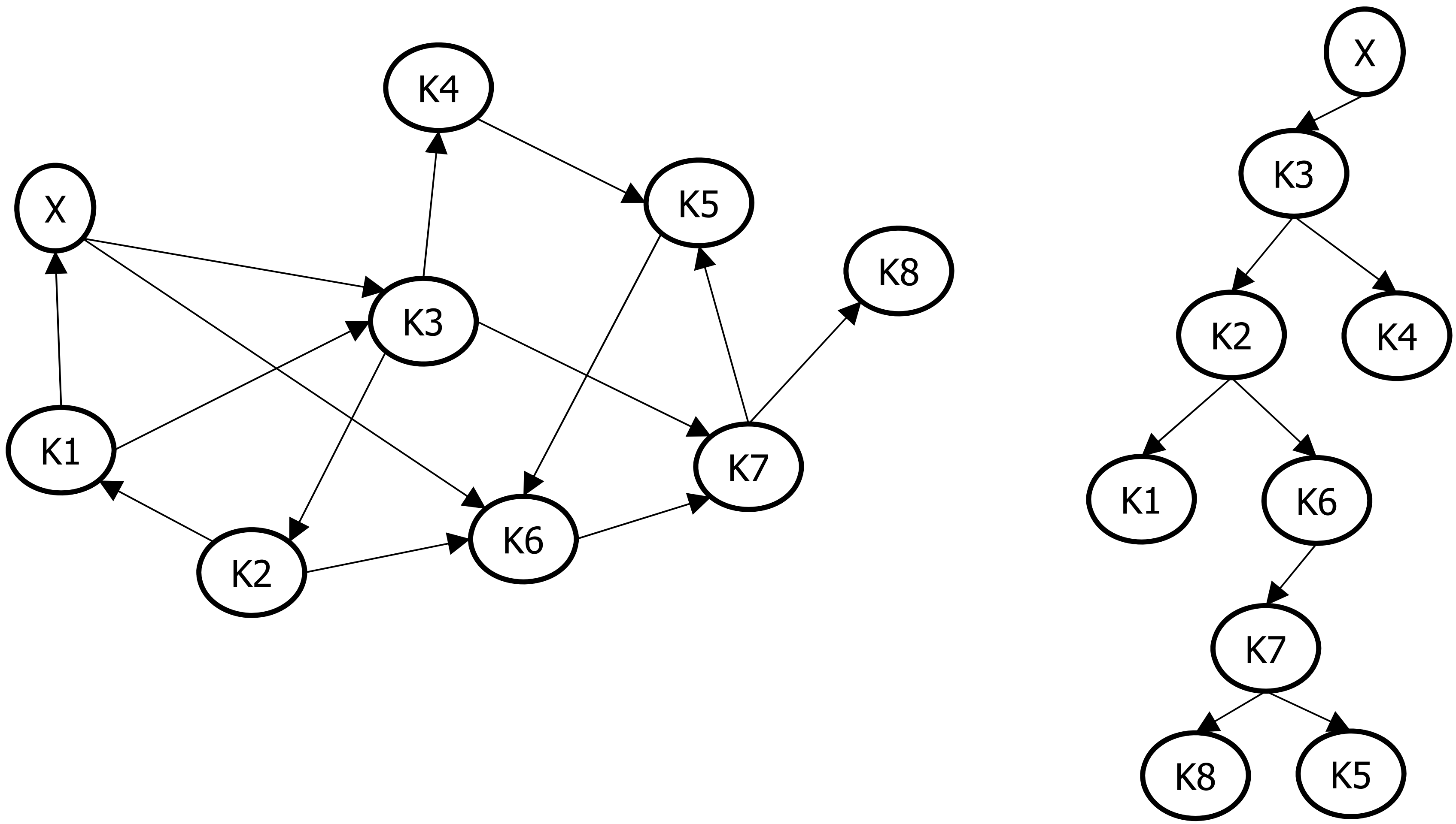


Most algorithms for finding SCCs are based on pre-/post-order labeling of nodes

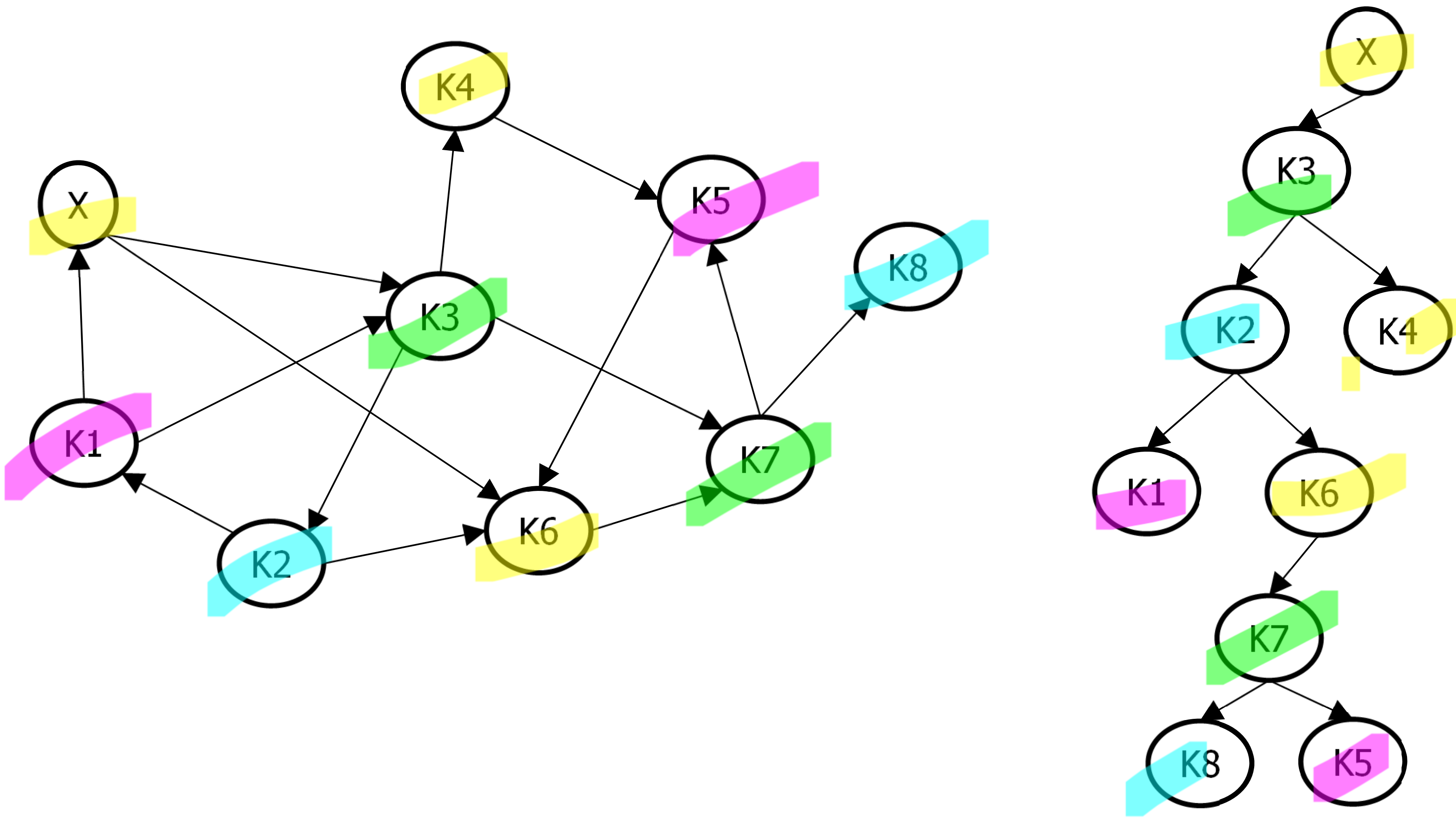
- Let G=(V, E). We assign each v∈V a pre-order and a post-order by the following method:
  - Init counters pre=post=0
  - Perform a depth-first traversal (DFS) of G, but use a modified, recursive traverse function
  - Whenever a node v is reached the first time, assign it the value of pre as pre-order value and increase pre
  - Whenever a node v is left the last time (all nodes below v traversed), assign it the value of post as post-order value and increase post

```
func void traverse (v node,
                    S,U list)
{
  pre += 1;
  pre(v) := pre;
  U := U \ {n};
  S := S ∪ {n};
  c := n.outgoingNodes();
  foreach x in c do
    if x∈U then
      traverse(v,S,U);
    end if;
  end for;
  post += 1;
  post(v) := post;
}
```

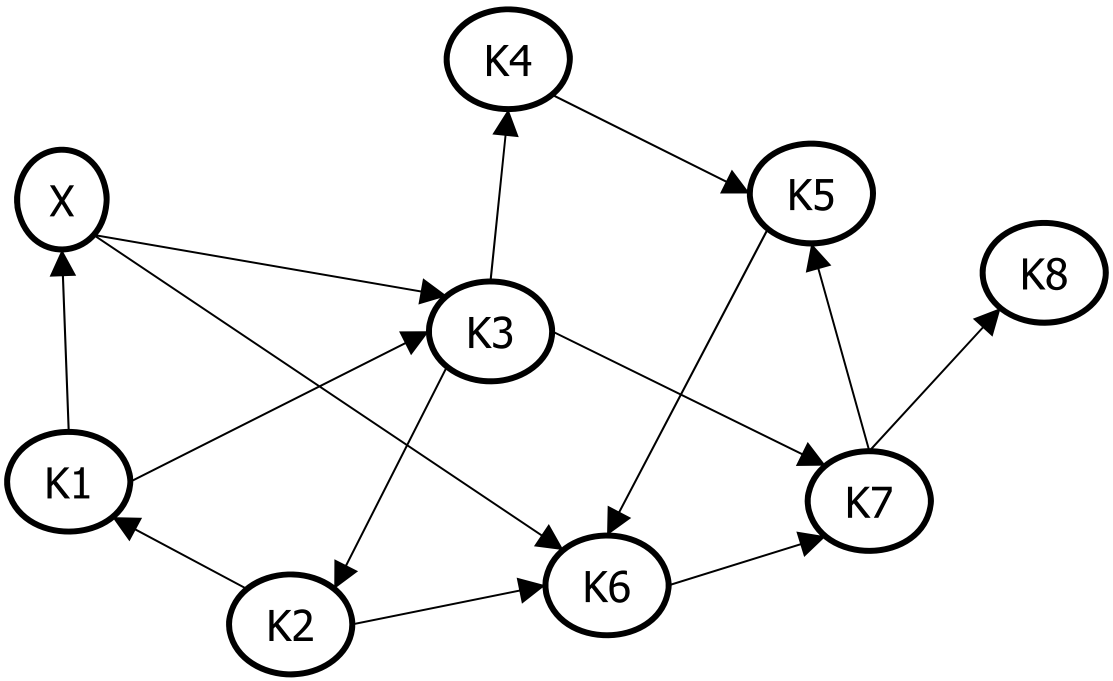
Example

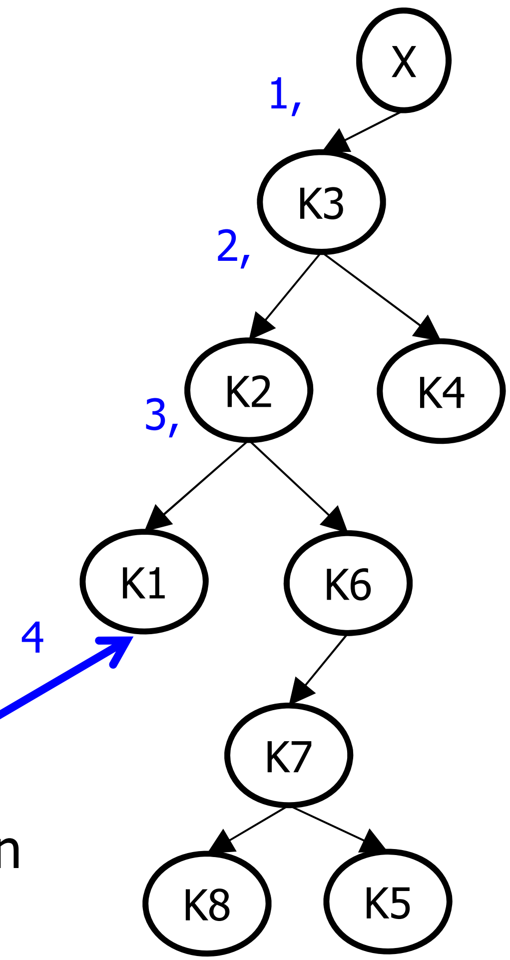


Example



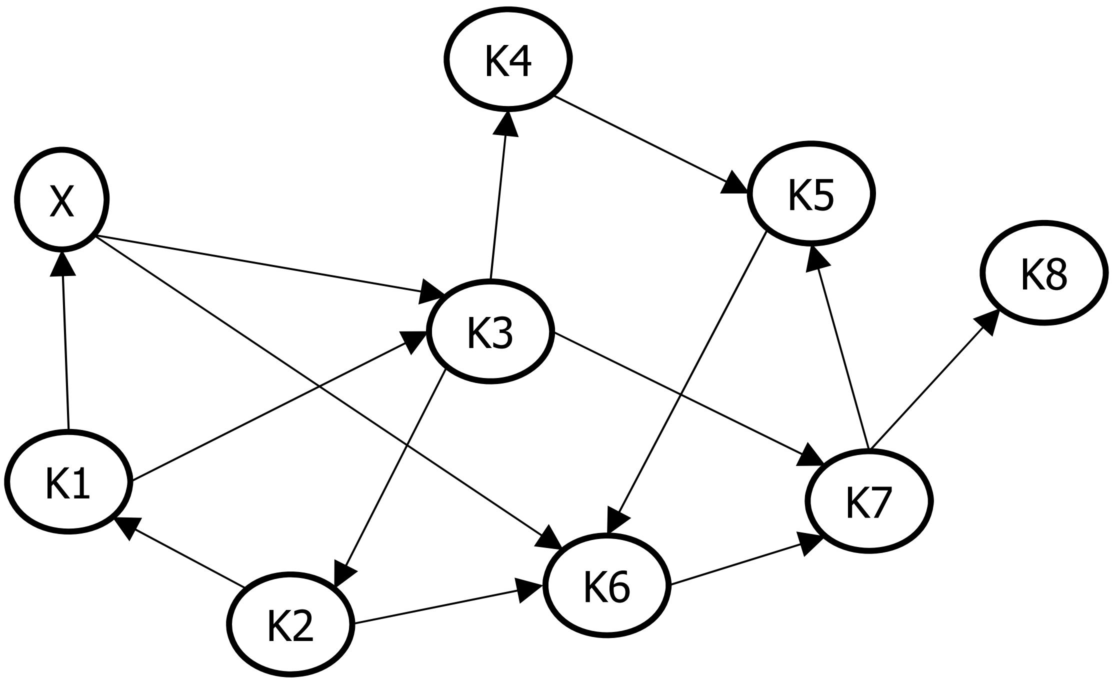
Example

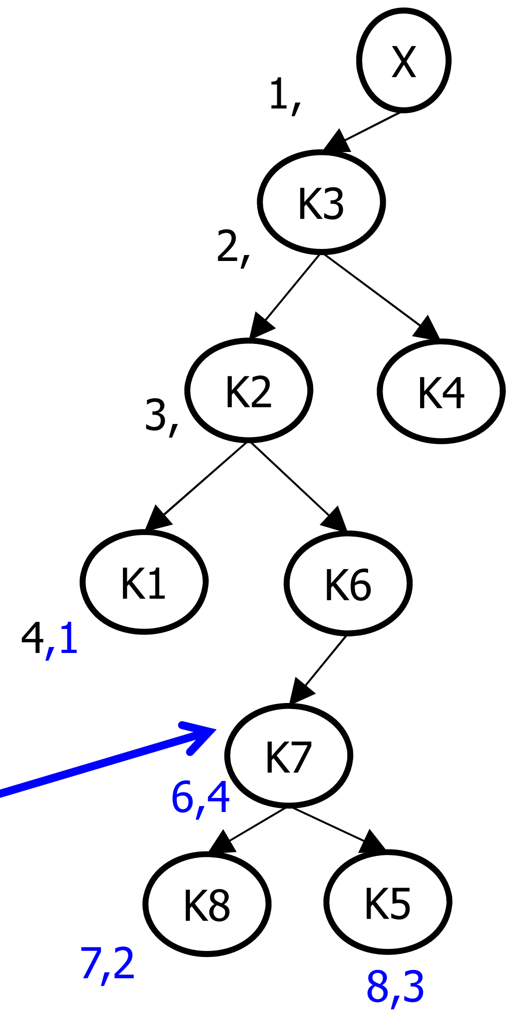




Last visit of K1: Could only go again to X, which was "seen" already; so set post(K1)=1

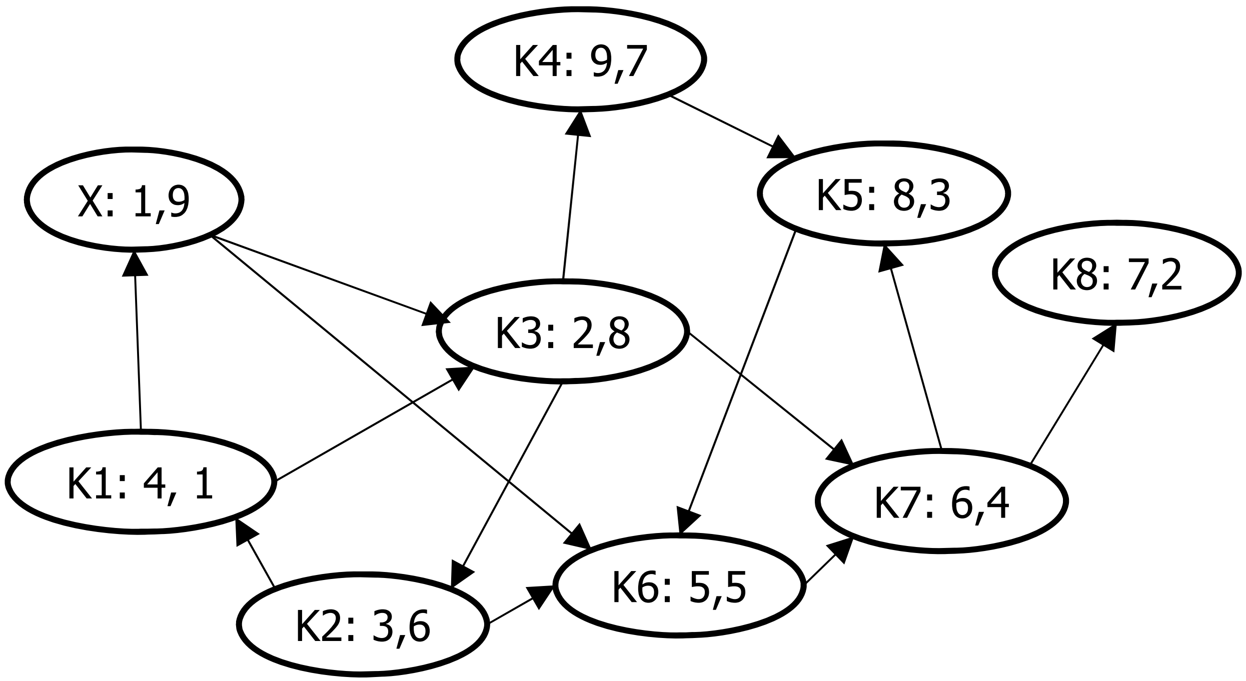
Example

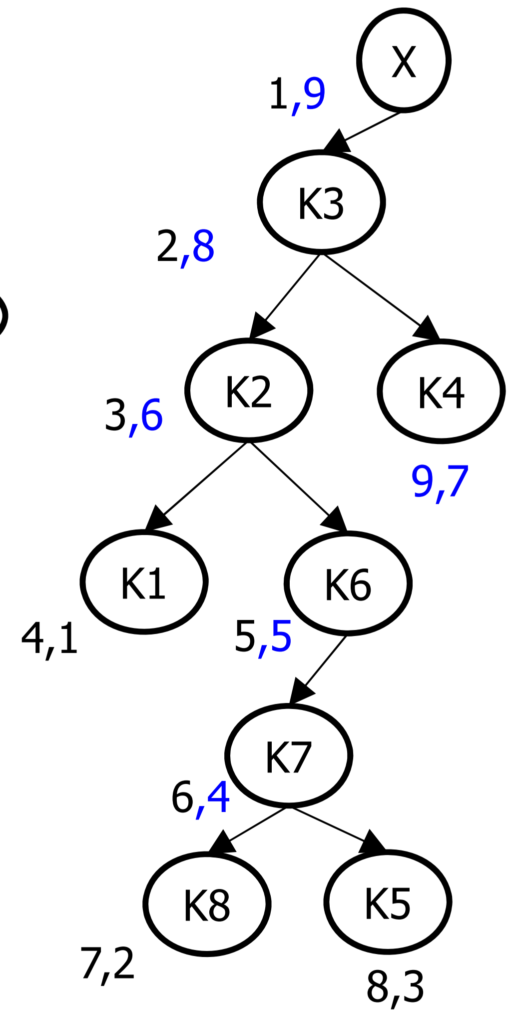




Last visit of K7: Could only go again to K5 and K8, which were "seen" already (whole subtree below K7 already explored); so set post(K7)=4

Example


K4: 9,7
K5: 8,3
X: 1,9
K8: 7,2
K3: 2,8
K1: 4, 1
K7: 6,4
K6: 5,5
K2: 3,6

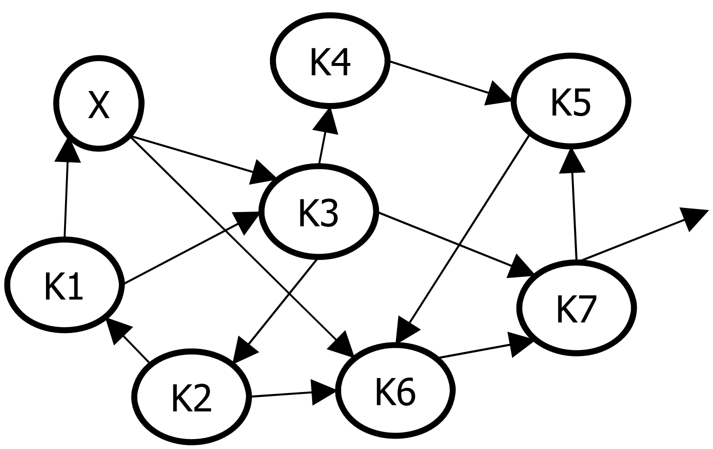

X
1,9
K3
2,8
K2
K4
3,6
9,7
K1
K6
4,1
5,5
K7
6,4
K8
K5
7,2
8,3

Content of this Lecture

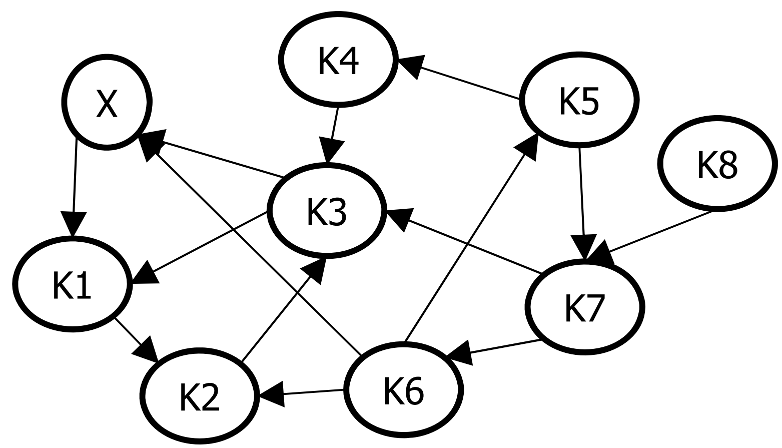
- Finding Connected Components in Undirected Graphs
- Finding Strongly Connected Components in Directed Graphs
  - Why?
  - Pre/Postorder Traversal
  - Kosaraju's algorithm

Kosaraju‵s Algorithm

- Definition:
  Let G=(V,E). The graph G^T=(V, E') with (v,w)∈E' iff (w,v) ∈E is called the *transposed graph* *of G.*



G



G^T

Kosaraju`s Algorithm

- Kosaraju's algorithm is very short
  - Compute post-order labels for all nodes from G using a first DFS
    - Here, we actually don't need the pre-order values
  - Compute G^T
  - Perform a second DFS on G^T always choosing as next node in the main loop of the FDS function the one with the highest post-order label according to the first DFS
  - All trees that emerge from the second DFS are SCC of G (and G^T)

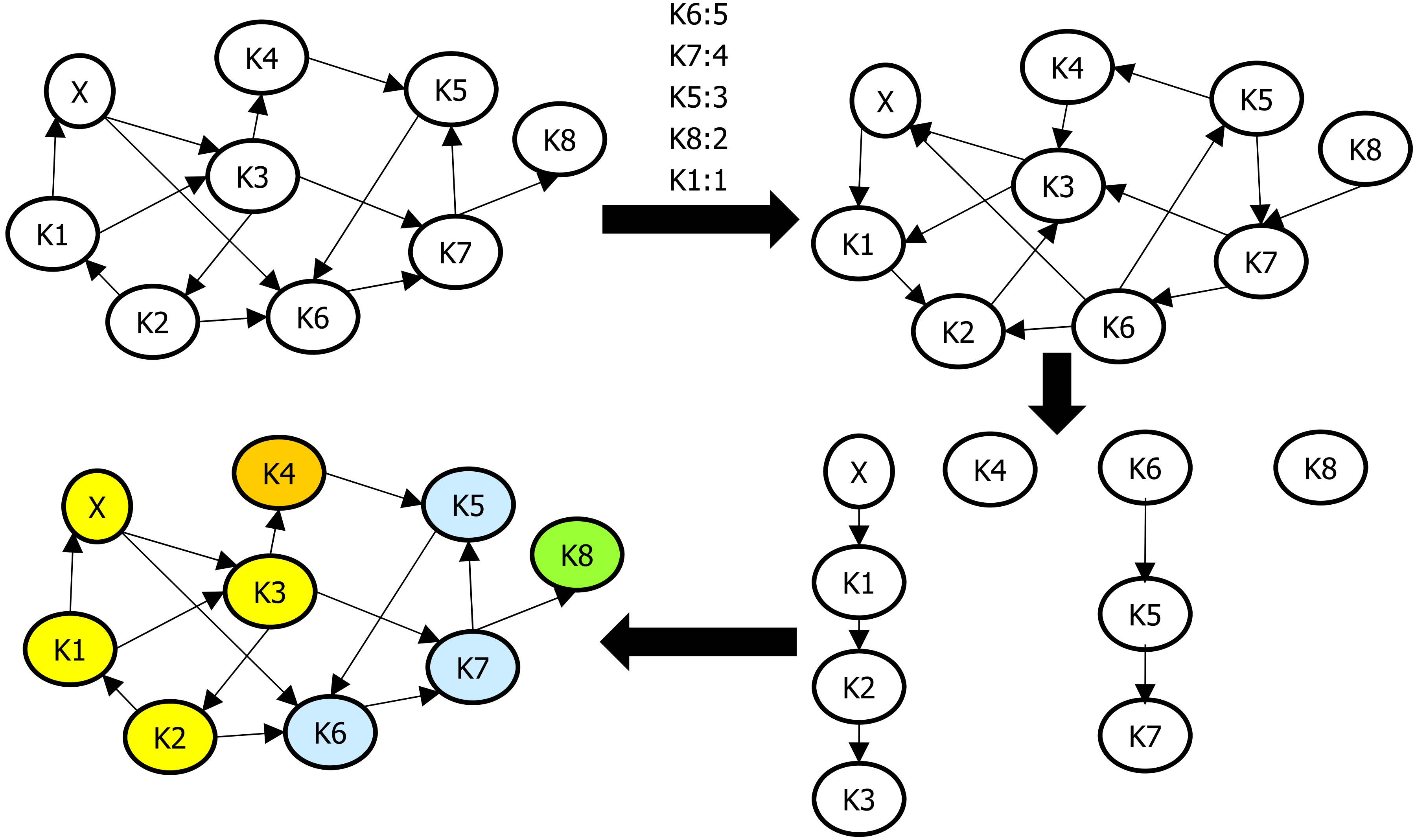
```
func void DFS ((V,E) graph) {
  U := V;          # Unseen nodes
  S := ∅;          # Seen nodes
  while U≠∅ do
    v := any_node_from(U);
    traverse( v, S, U);
  end while;
}
```

→

```
func void DFS ((V,E) graph) {
  U := V;          # Unseen nodes
  S := ∅;          # Seen nodes
  while U≠∅ do
    v := highest_post_order_node_from(U);
    traverse( v, S, U);
  end while;
}
```

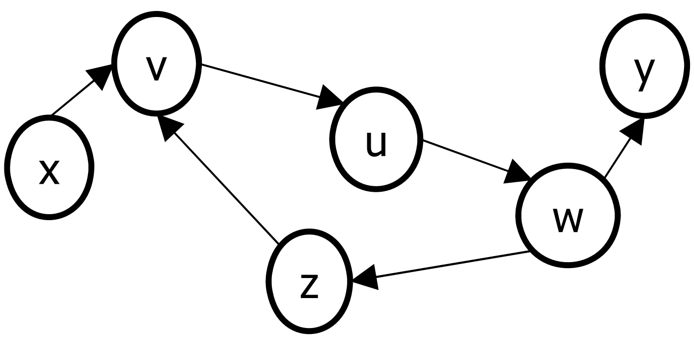
Example

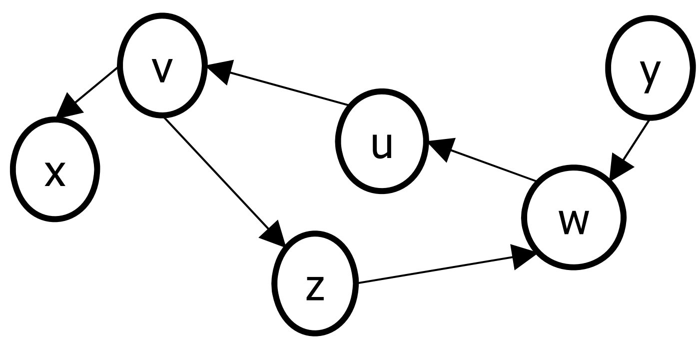
X:9
K3:8
K4:7
K2:6
K6:5
K7:4
K5:3
K8:2
K1:1



Correctness

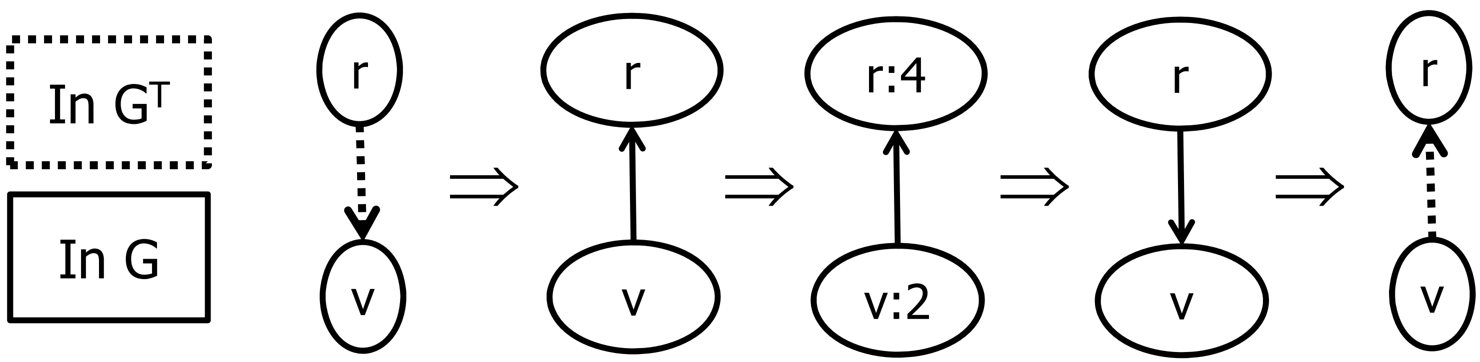
- We prove that two nodes v, w are in the same tree of the second DFS iff v and w are in the same SCC in G
- Proof
  - $\Leftarrow$: Suppose v$\rightarrow$w and w$\rightarrow$v in G. One of the two nodes (assume it is v) must be reached first during the second DFS. Since v can be reached by w in G, w can be reached by v in G^T. Thus, when we reach v during the traversal of G^T, we will also reach w further down the same tree, so they are in the same tree of G^T.



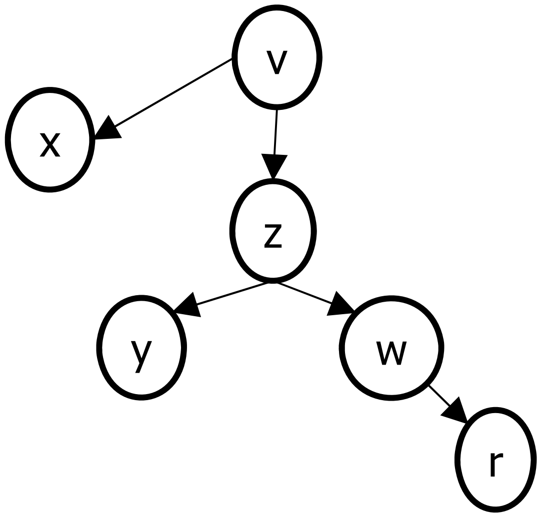


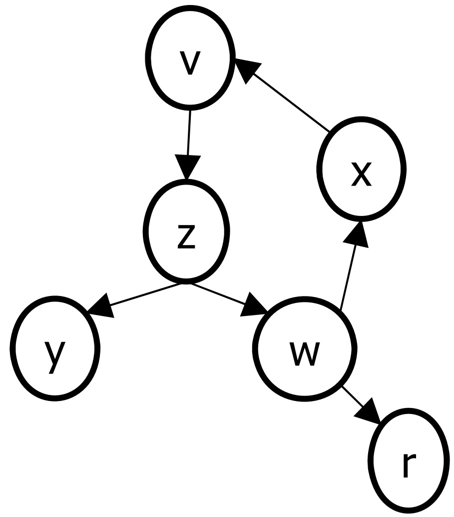
Correctness

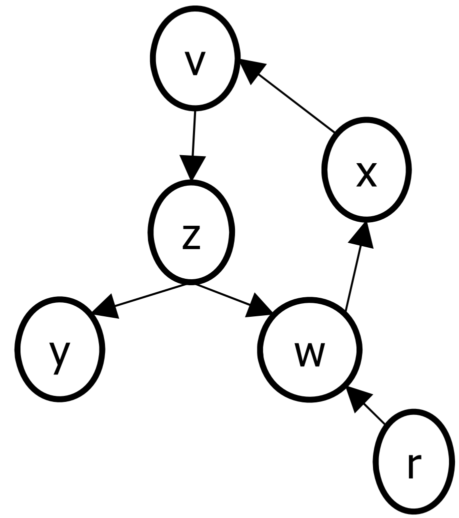
- $\Rightarrow$: Suppose v and w are in the same DFS-tree of G^T
  - Suppose r is the root of this tree
  - Since $r \rightarrow v$ in G^T, it must hold that $v \rightarrow r$ in G
  - Because of the order of the second DFS: post(r)>post(v) in G
  - Thus, there must be a path $r \rightarrow v$ in G: Otherwise, last visit of r had been before v in G and thus r would have a smaller post-order
  - Since $v \rightarrow r$ and $r \rightarrow v$ in G, the same is true for G^T
  - The same argument shows that $w \rightarrow r$ and $r \rightarrow w$ in G
  - By transitivity, it follows that $v \rightarrow w$ and $w \rightarrow v$ via r in G (and in G^T)



Examples (p() = post-order())







- $v \to w$
- Thus, $w \to v$ in G^T
- Because $w \nrightarrow v$ in G, p(v)>p(w)
- First tree in G^T starts in v; doesn't reach w
- v, w not in same tree

- $v \to w$ and $w \to v$ in G and in G^T
- Assume w is first in 1st DFS: p(w)>p(v)
- w has higher p-value, thus 2^{nd} DFS starts in w and reaches v
- v, w in same tree

- Let's start 1^{st} DFS in r: p(r)>p(w)>p(v)
- 2^{nd} DFS starts in r, but doesn't reach w
- Second tree in 2^{nd} DFS starts in w and reaches v
- v, w in same tree

Complexity

- Both DFS are in O(m+n), computing G^T is in O(m)
- Instead of computing post-order values and sort them, we can simple push nodes on a stack when we leave them the last time – needs to be done O(n) times
- Together: O(m+n)
  - Needs one more array to remove selected nodes during second DFS from stack in constant time
- Since in WC we need to look at each edge and node at least once to find SCCs, the problem is in Ω(m+n)
- There are faster algorithms that find SCCs in one traversal
  - Tarjan's algorithm, Gabow's algorithm