Algorithms and Data Structures

Strongly Connected Components

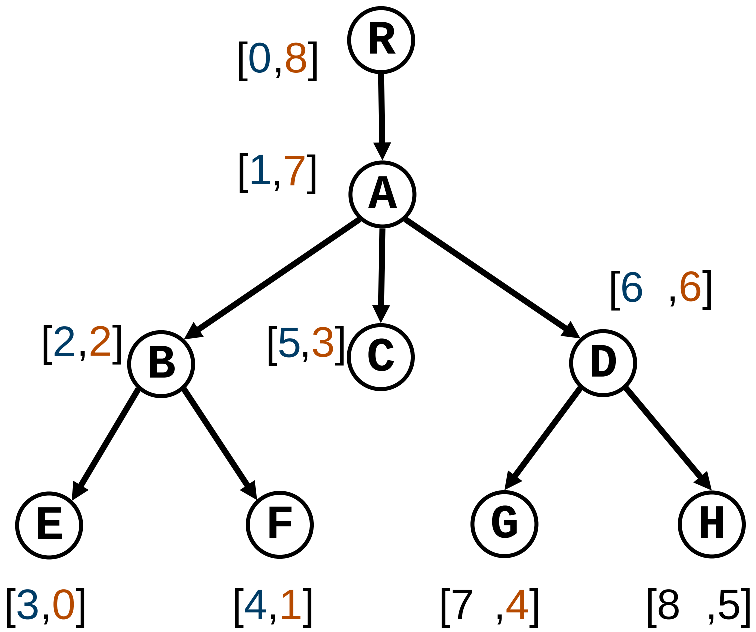
Ulf Leser

Content of this Lecture

- Graph Traversals
- Strongly Connected Components

Recall: Reachability in Trees

- Assume a DFS-traversal
- Build an array assigning each node two numbers
- Preorder numbers
  - Keep a counter pre
  - Whenever a node is entered the first time, assign it the current value of pre and increment pre
- Postorder numbers
  - Keep a counter post
  - Whenever a node is left the last time, assign it the current value of post and increment post



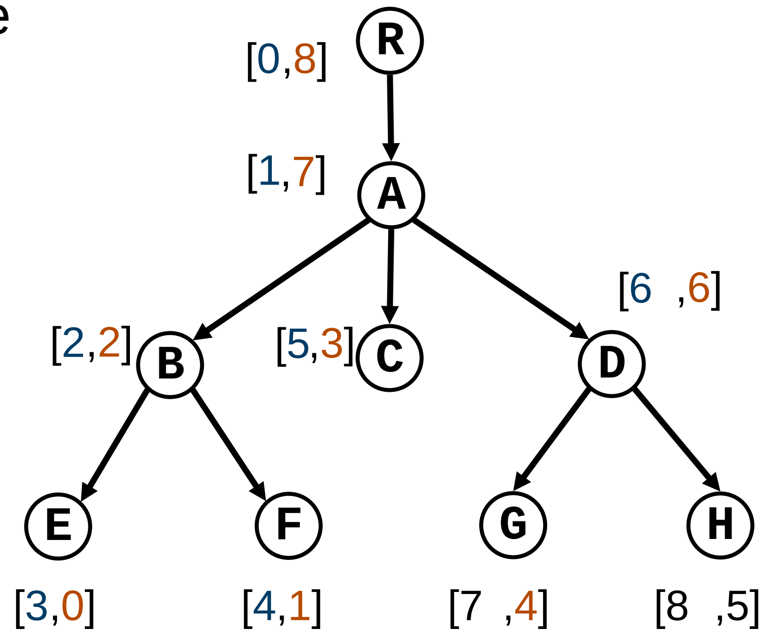
Examples from S. Trissl, 2007

Ancestry and Pre-/Postorder Numbers

- Trick: A node v is reachable from a node w iff

$$pre(v)>pre(w) \wedge post(v)<post(w)$$

- Explanation
  - v can only be reached from w, if w is "higher" in the tree, i.e., v was traversed after w and hence has a higher preorder number
  - v can only be reached from w, if v is "lower" in the tree, i.e., v was left before w and hence has a lower postorder number
- Analysis: Test is O(1)



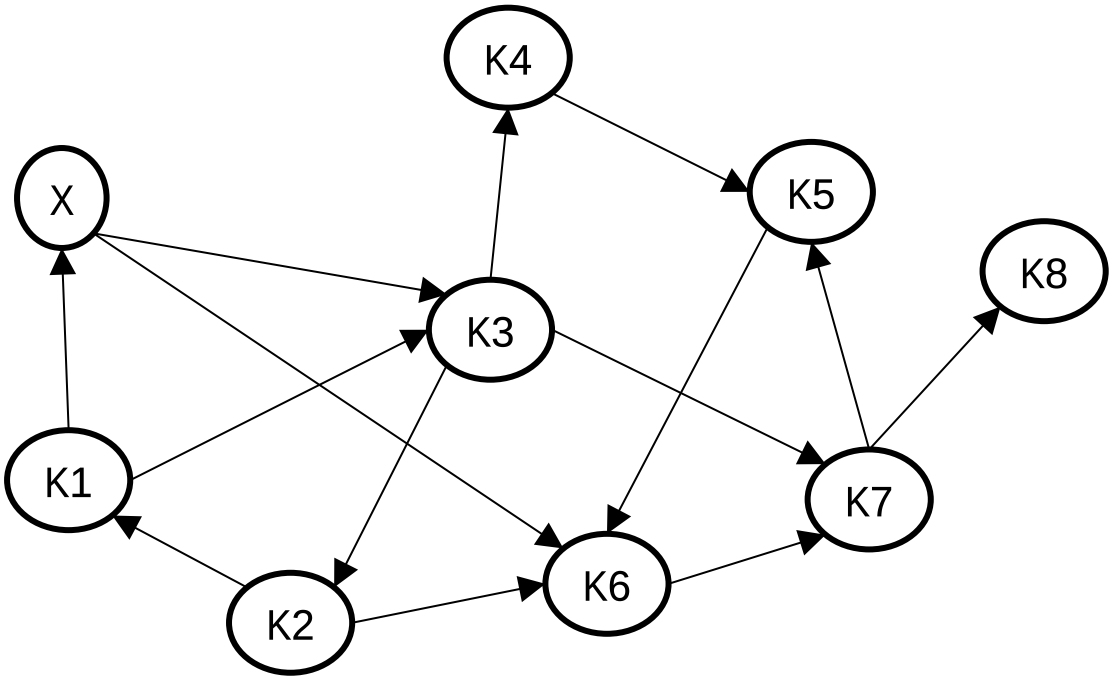
Pre-/Post-order Labeling for Graphs

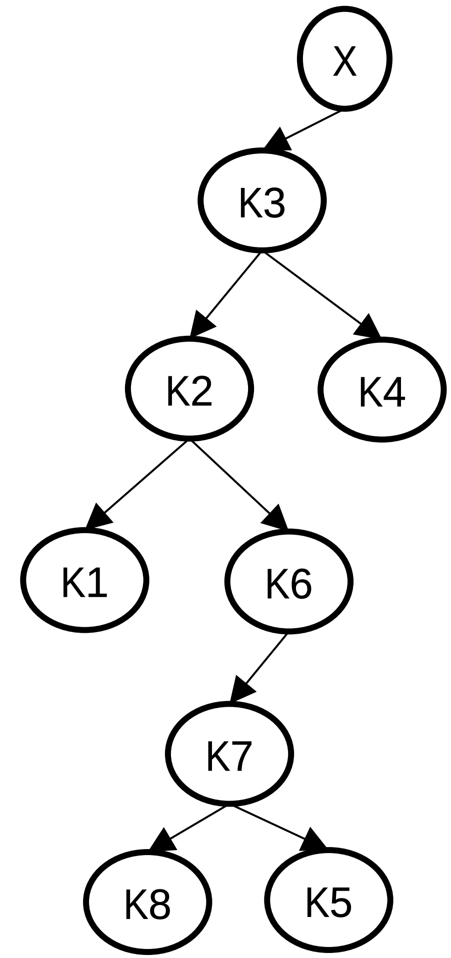
- Method

  Let G=(V, E). We assign each v∈V a pre-order and a post-order as follows. Set pre=post=1. Perform a depth-first traversal of G, starting at arbitrary nodes. When 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, assign it the value of post as post-order value and increase post.

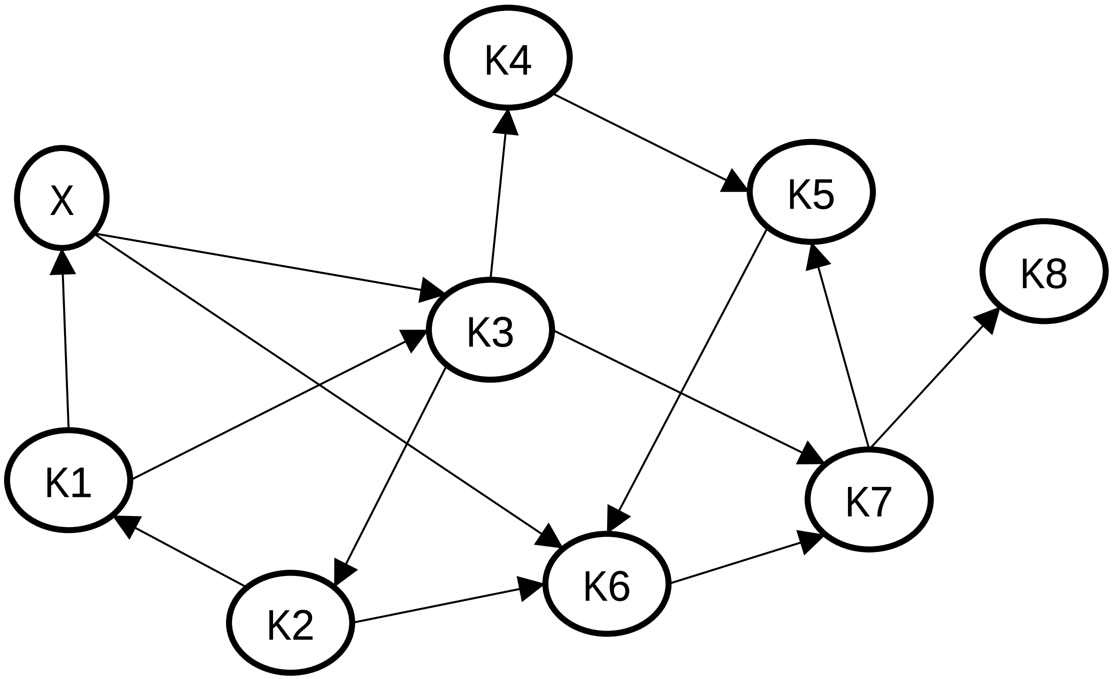
- Notes
  - Traversals are cycle-free by definition –avoid multiple visits
  - Complexity: O(|V|+|E|))
  - Labeling not unique; depends on chosen start nodes and order in which children are visited

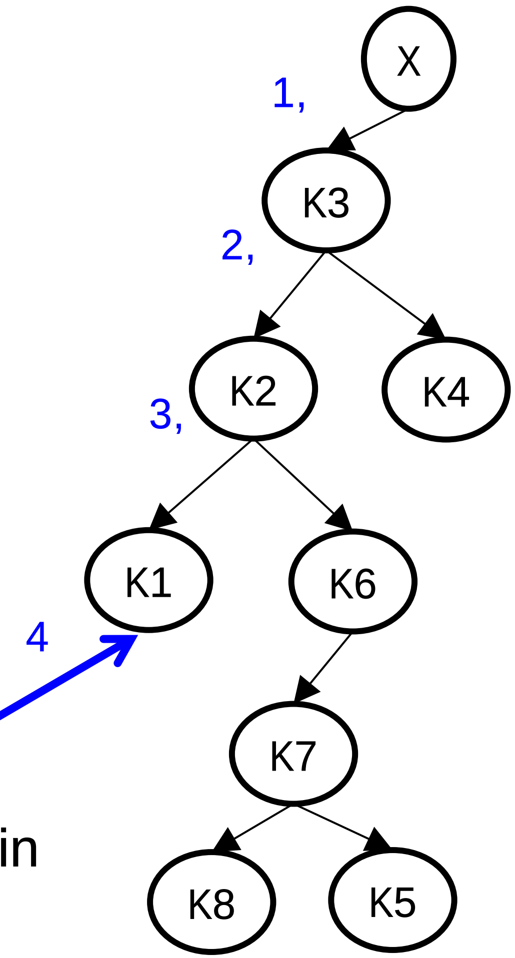
Example





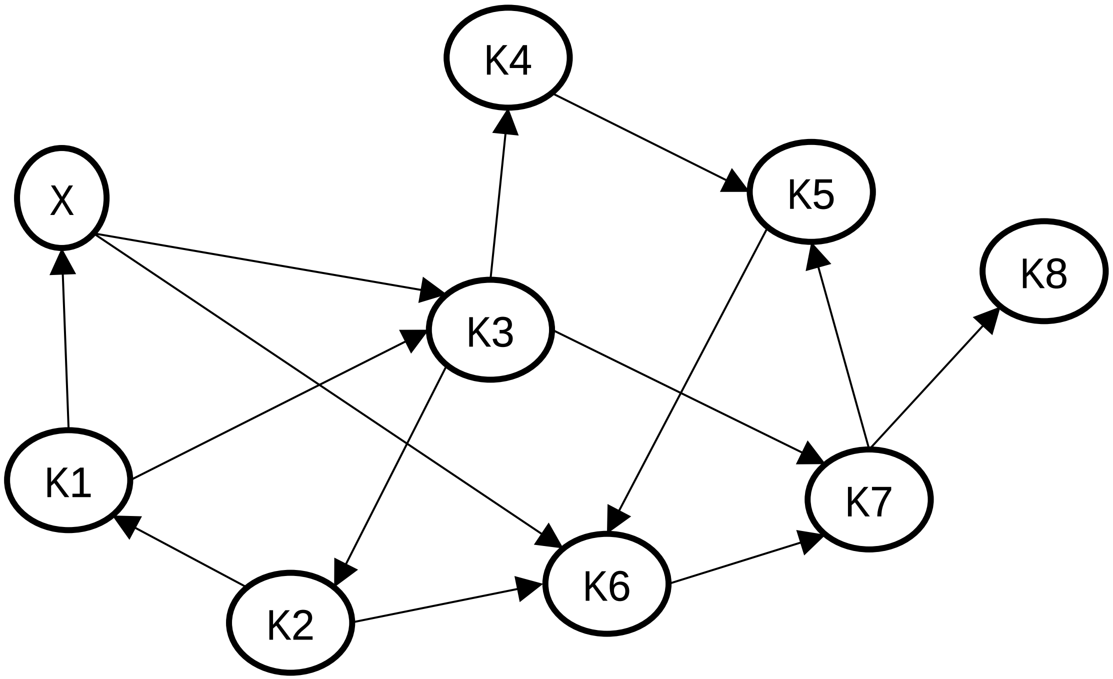
Example

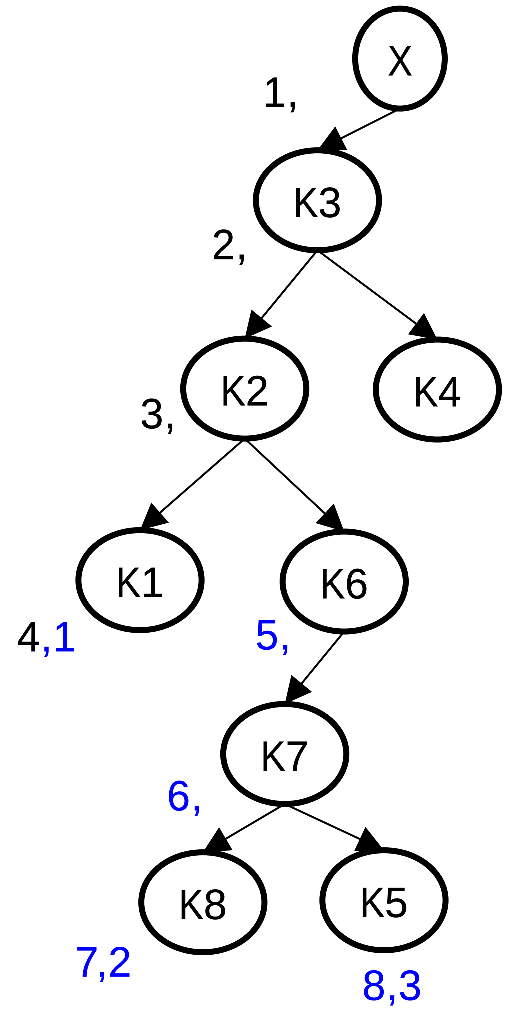




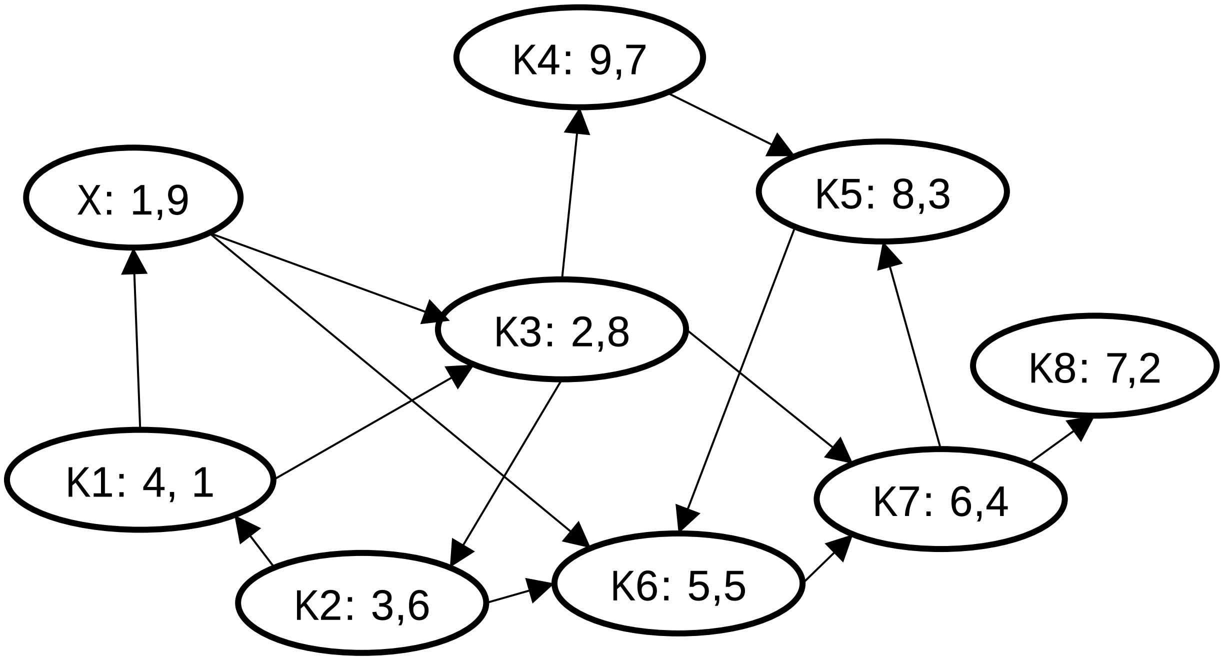
Last visit: Cannot be visited again without running into a cycle

Example

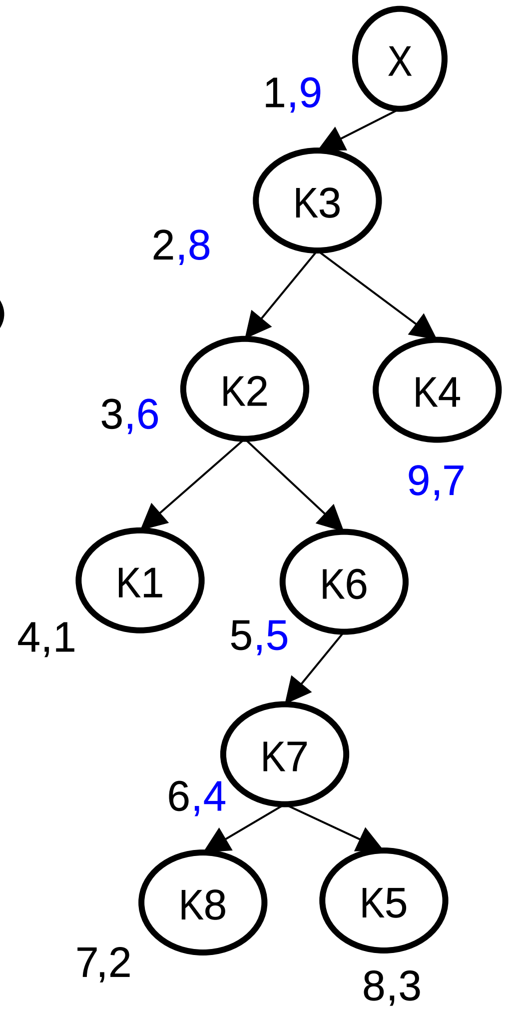




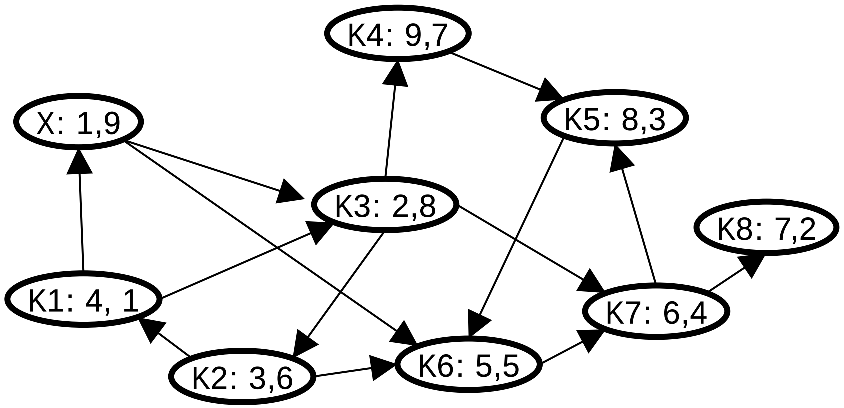
Example



- Reachability trick does not work
- Example: K1-K4
  - Reachable in G
  - But pre(K4)>pre(K1)

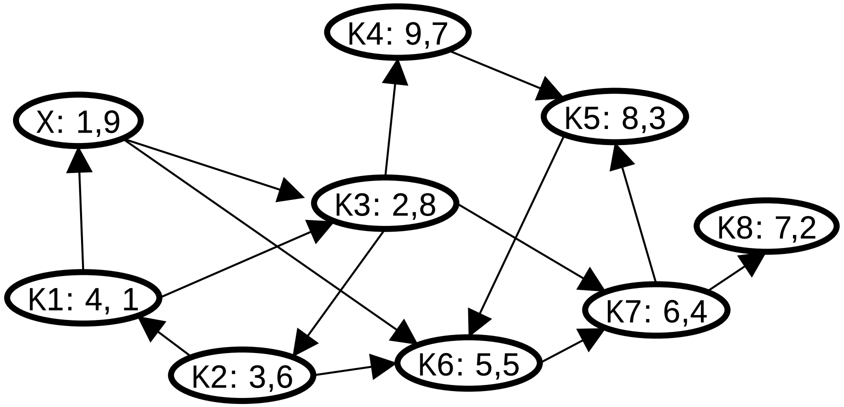


Tricks to Speed-Up Reachability in Graphs



- Much research over the last decade
  - PPO: Pre-/Post-Order Pair
- Ideas
  - If the graph is “tree-like” and acyclic
  - Follow all paths and assign multiple PPOs
  - Requires exponential space in WC, depending on “tree-likeliness”

Tricks to Speed-Up Reachability in Graphs

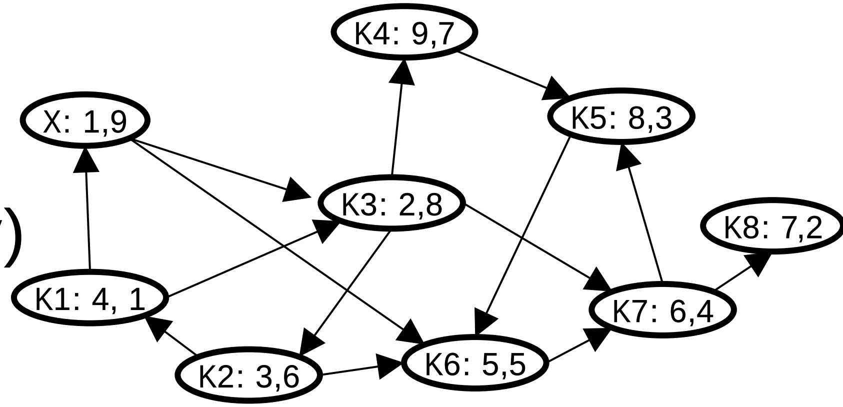


- Ideas (GRIPP)
  - If the graph is acyclic
  - Perform a modified DFS
    - When a node is visited for the none-first time, assign another PPO but to not continue traversal further
    - For each node, store all PPOs
  - During search, expand with nodes which have multiple PPOs
    - Expand: “Jump” to the first PPO and branch another search
  - “Almost constant” runtime in many graphs

Trissl, S. and Leser, U. (2007). "Fast and Practical Indexing and Querying of Very Large Graphs". SIGMOD.

Tricks to Speed-Up Reachability in Graphs

- Observation: If v is reachable from w, then there exists a DFS of G in which pre(w)<pre(v) and post(w)>post(v)
  - Example K1-K4: Start DFS in K1



- Idea
  - Perform a fixed number (k) of DFS and use as filter
  - If v is reachable from w in any of the DFS: Done.
  - Otherwise use another method (hopefully not often!)
  - Very effective in dense graphs where most nodes are reachable
  - Parameter k controls runtime and space

Yildirim, H., Chaoji, V. and Zaki, M. J. (2010). "GRAIL: Scalable Reachability Index for Large Graphs." *VLDB*

Graph Transformations

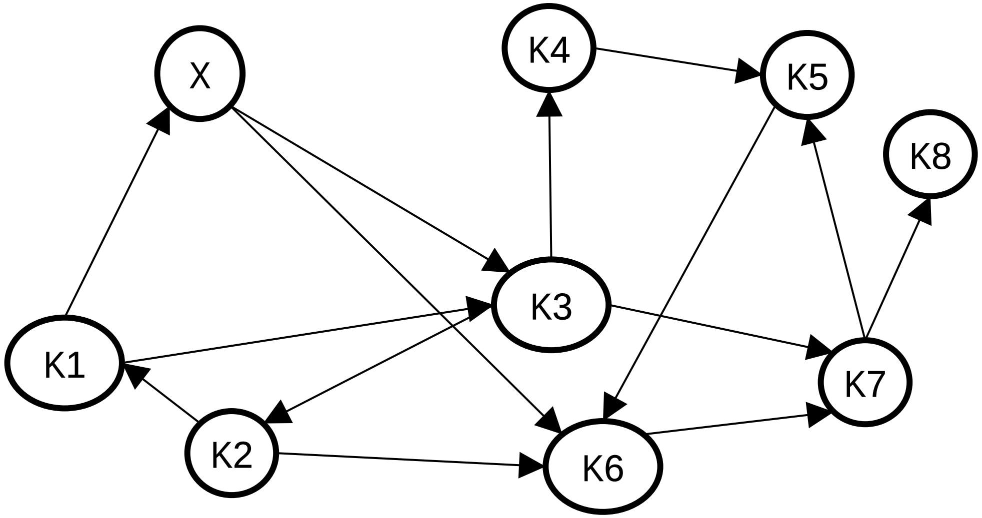
- Many other suggestions
- All require a preprocessing phase (e.g. PPO indexing) and a search phase
- Complexities of both phases depend fundamentally on |G|
  - If we could shrink G (without losing reachability-relevant information), all algorithms would be much faster
- Furthermore, some methods only work with acyclic graphs
  - We need a way to transform a cyclic graph G into an acyclic graph G' which encoded the same reachability information

Content of this Lecture

- Graph Traversals
- Strongly Connected Components (SCC)
  - Motivation: Graph Contraction
  - Kosaraju's algorithm

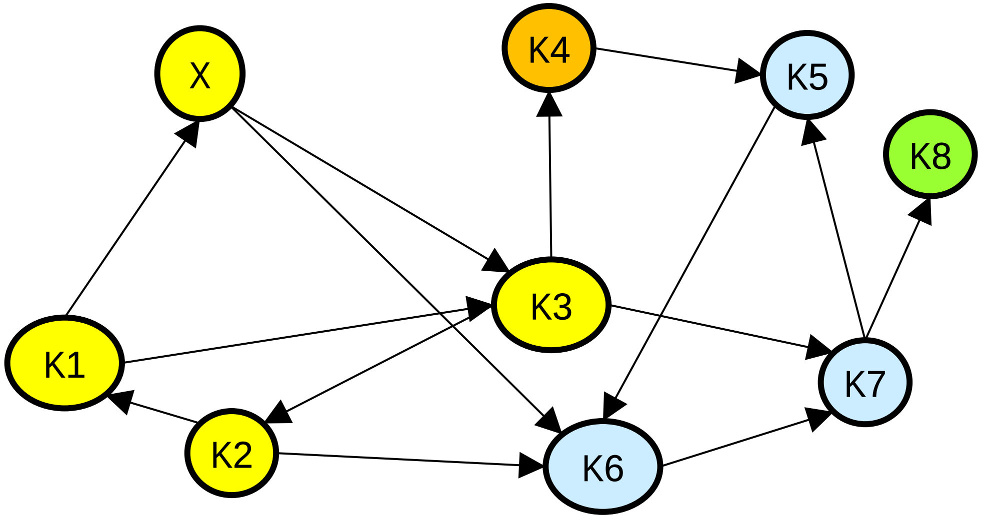
Recall

- Definition
  Let G=(V, E) be a directed graph.
  - *An induced subgraph G'=(V', E') of G is called connected if G' contains a path between any pair v,v'∈V'*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



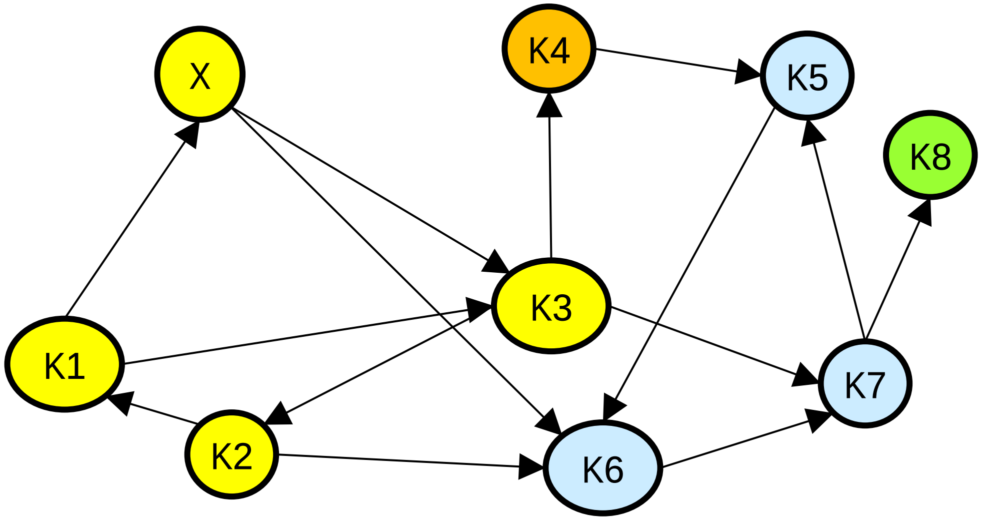
Recall

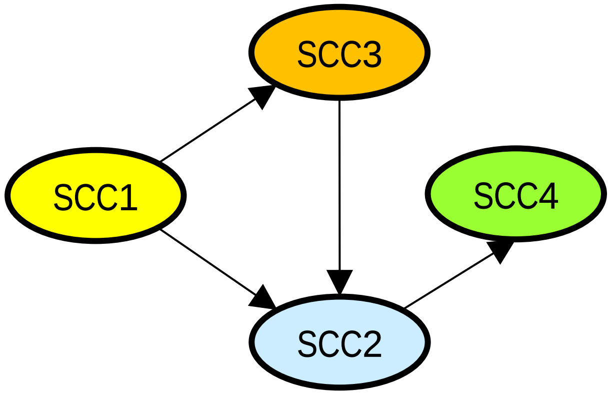
- Definition
  Let G=(V, E) be a directed graph.
  - *An induced subgraph G'=(V', E') of G is called connected if G' contains a path between any pair v,v'∈V'*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



Motivation: 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





Reachability and Graph Contraction

- Intuitively: TC(G) = TC(G')+SCC(G)
  - Representing SCC(G): Hash table h mapping each node ID to its SCC-ID
  - Testing reachability $v \rightarrow w$: Test if h(v)=h(w)
  - Thus, we only have to consider G' further
- Computing SCC solves our problems in graph reachability
  - "If we could shrink G (without losing reachability-relevant information), all algorithms would be much faster"
    - Yes we can
  - "We need a way to transform a cyclic graph G into an acyclic graph G' which encoded the same reachability information"
    - Yes we can
- But – how much work do we need to compute SCC(G)?

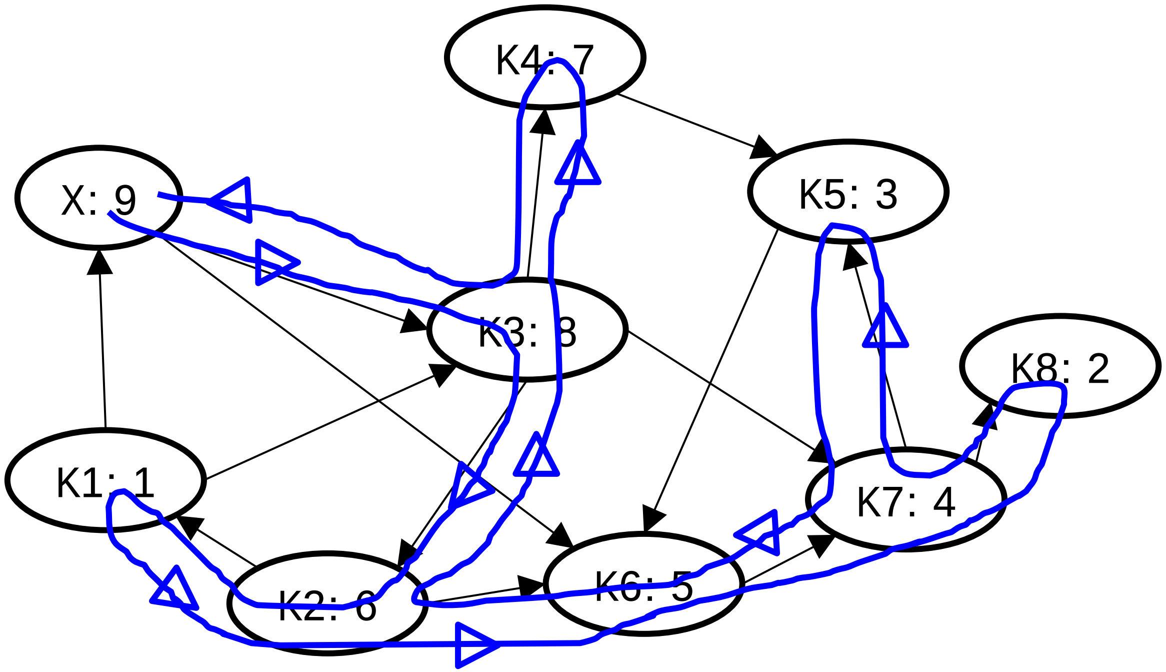
Content of this Lecture

- Graph Traversals
- Strongly Connected Components (SCC)
  - Motivation
  - Kosaraju's algorithm

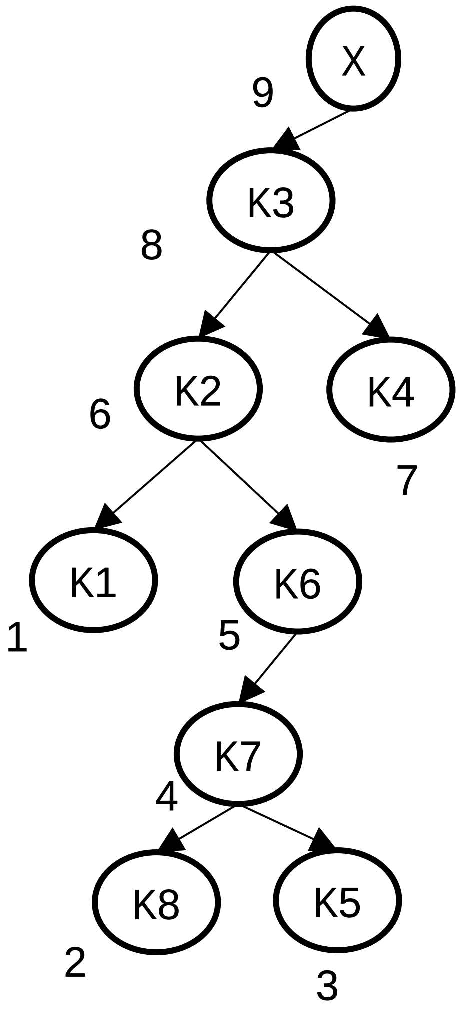
Kosaraju‘s Algorithm

- Definition
  Let $G=(V,E)$. The graph $G^T=(V, E')$ with $(v,w) \in E'$ iff $(w,v) \in E$ is called the transposed graph of G.
- Kosaraju’s algorithm is very short (but not simple)
  - Compute post-order labels for all nodes from G using a first DFS
    - We don’t need pre-order values
  - Compute G^T
  - Perform a second DFS on G^T always choosing as next node 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)
- Unpublished; Kosaraju, 1978

Example



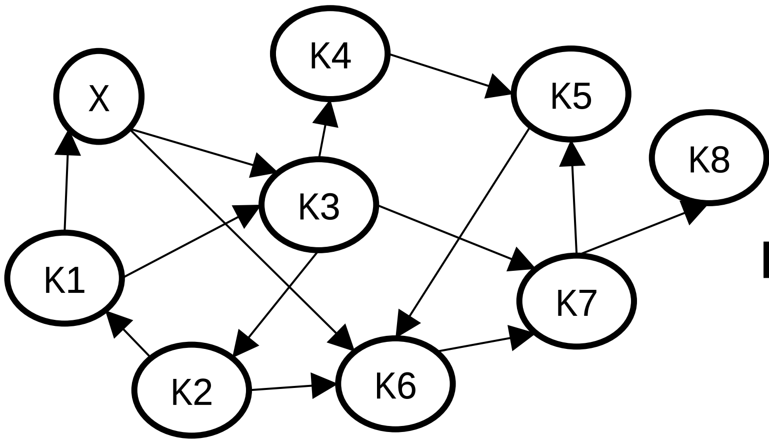
K4: 7
X: 9
K5: 3
K3: 8
K8: 2
K1: 1
K7: 4
K2: 6
K6: 5

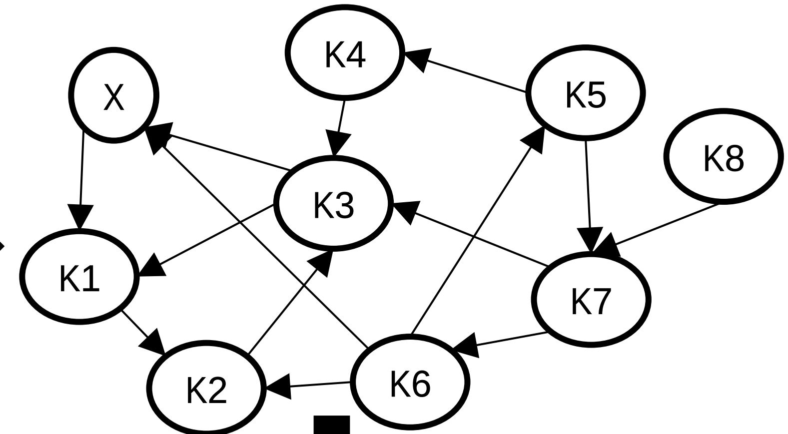


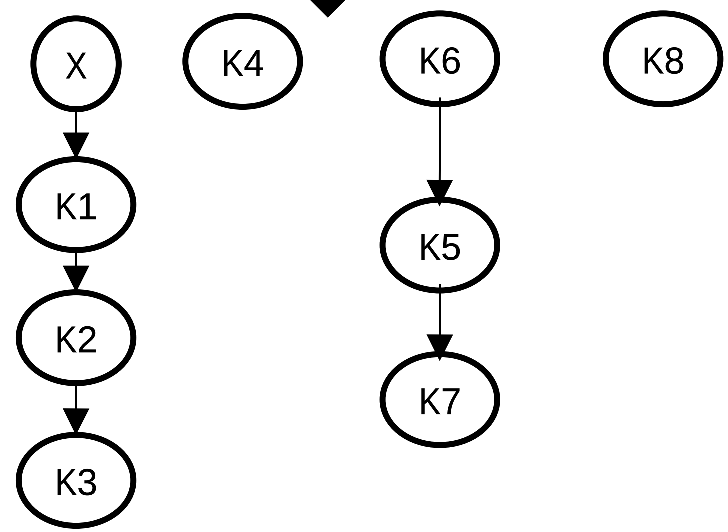
X
9
K3
8
K2
K4
6
7
K1
K6
1
5
K7
4
K8
K5
2
3

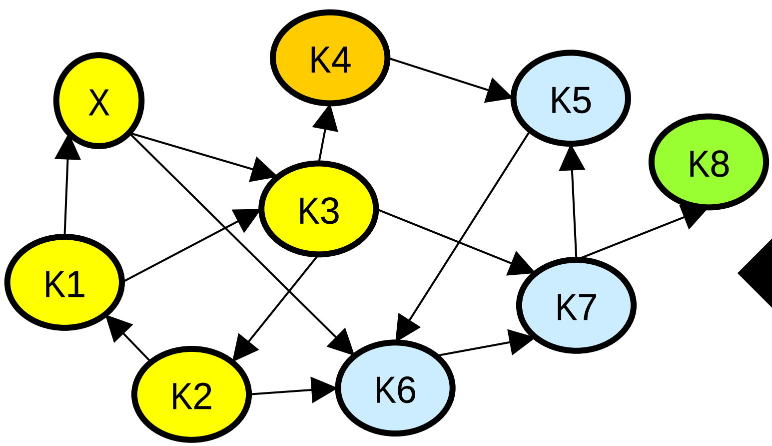
Example

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



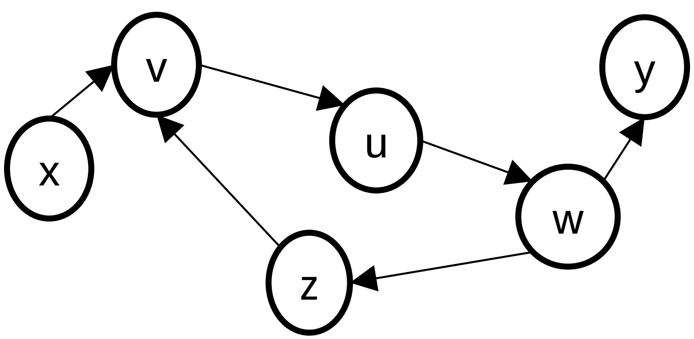


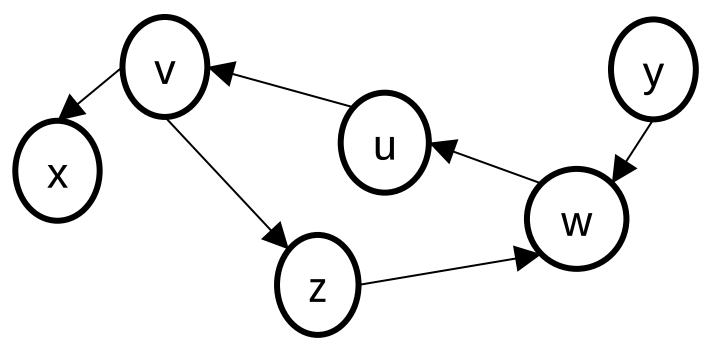




Correctness

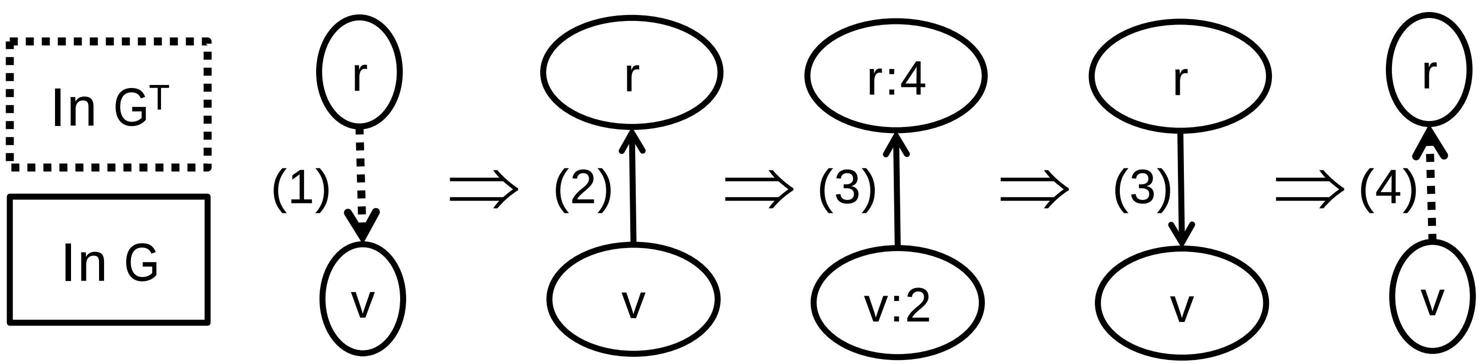
- Theorem
  Let G=(V,E). *Any two nodes v, w are in the same tree of the second DFS iff v and w are in the same SCC in G.*
- Proof
  - ⇐: Suppose v→w and w→v in G. One of the two nodes (assume it is v) must be reached first during the second DFS. Since v can be reached by w in G, w can be reached by 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.



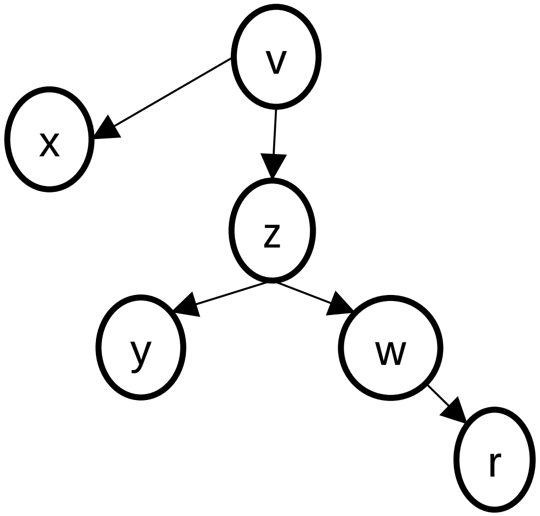


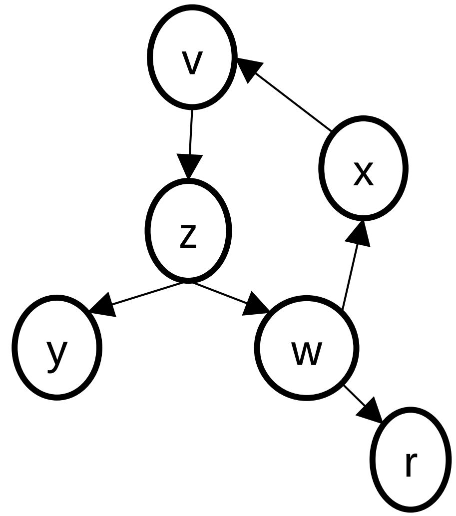
Correctness

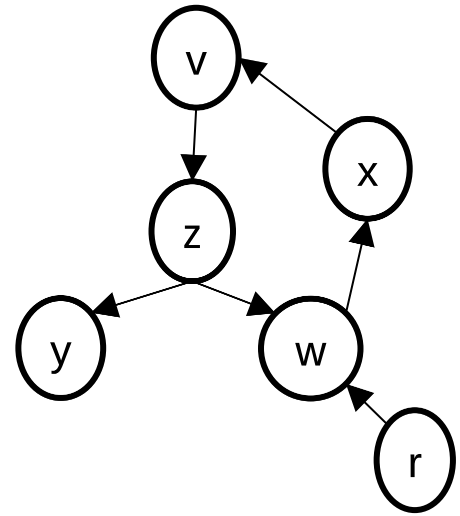
- ⇒: Suppose v and w are in the same DFS-tree of G^T
  - Suppose r is the root of this tree
  - (1) Since r→v in G^T, it must hold that v→r in G
  - (2) Because of the order of the second DFS: post(r)>post(v) in G
  - (3) Thus, there must be a path r→v in G: Otherwise, r had been visited last after v in G and thus would have a smaller post-order
  - (4) Since v→r (1) and r→v (3) in G, the same is true for G^T
  - (5) The same argument shows that w→r and r→w in G
  - (6) By transitivity, it follows that v→w and w→v via r in G and in G^T



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







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

- $v \rightarrow w$ and $w \rightarrow v$ in G and in G^T
- Assume w is first in 1st DFS: $p(w)>p(v)$
- Thus 2nd DFS starts in w and reaches v
- v, w in same tree

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

Complexity

- Both DFS are in O(|G|), computing G^T is in O(|E|)
- Instead of computing post-order values and sort them, we can simple push nodes on a stack when we leave them the last time in the first DFS – needs to be done O(|V|) times
- In the 2nd DFS, we pop nodes from the stack as new roots
  - Needs one more array to remove selected nodes during second DFS from stack in constant time
- Together: O(|V|+|E|)
  - Optimal: Since in WC we need to look at each edge and node at least once to find SCCs, the problem is in Ω(|V|+|E|)
- There are faster algorithms that find SCCs in one traversal
  - Tarjan's algorithm, Gabow's algorithm