Algorithms and Data Structures

Strongly Connected Components

Ulf Leser

Content of this Lecture

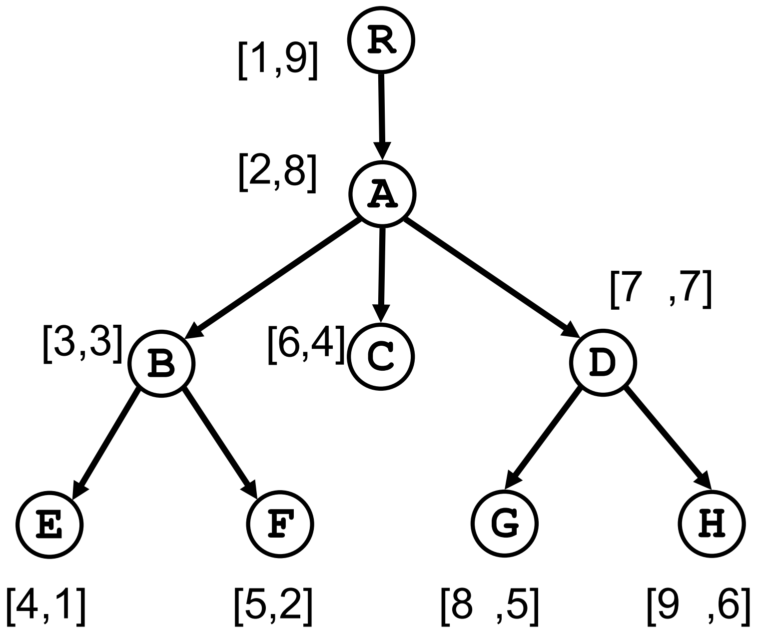
- Graph Traversals
- Strongly Connected Components

Reachability in Graphs

- Fundamental problem: Given a graph G=(V,E) and a pair of nodes $v,w \in V$: Is w reachable from v?
- Solutions so far ($n=|V|$)
  - Warshall's algorithm solves the problem for all pairs, but $O(n^3)$
  - Dijkstra solves the problem for a given pair, but $O(n^2)$
    - Our implementation was $O(n^2*\log(n))$
    - Actually finds the shortest path, which we don't need
- Can we do better?
  - Yes: By pre-processing the graph (graph indexing)

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`**



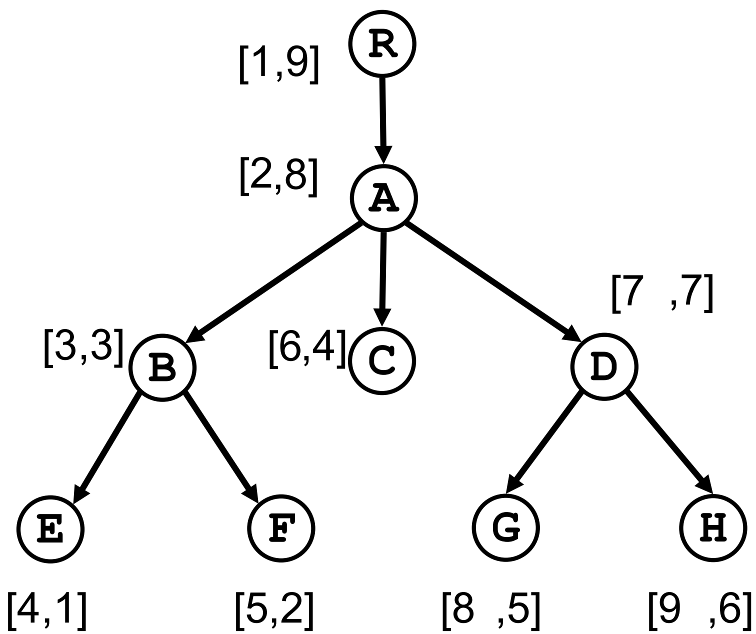
Examples from S. Trissl, 2007

Ancestry and Pre-/Postorder Numbers

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

  $$\mathtt{pre(v)>pre(w)} \wedge \mathtt{post(v)<post(w)}$$

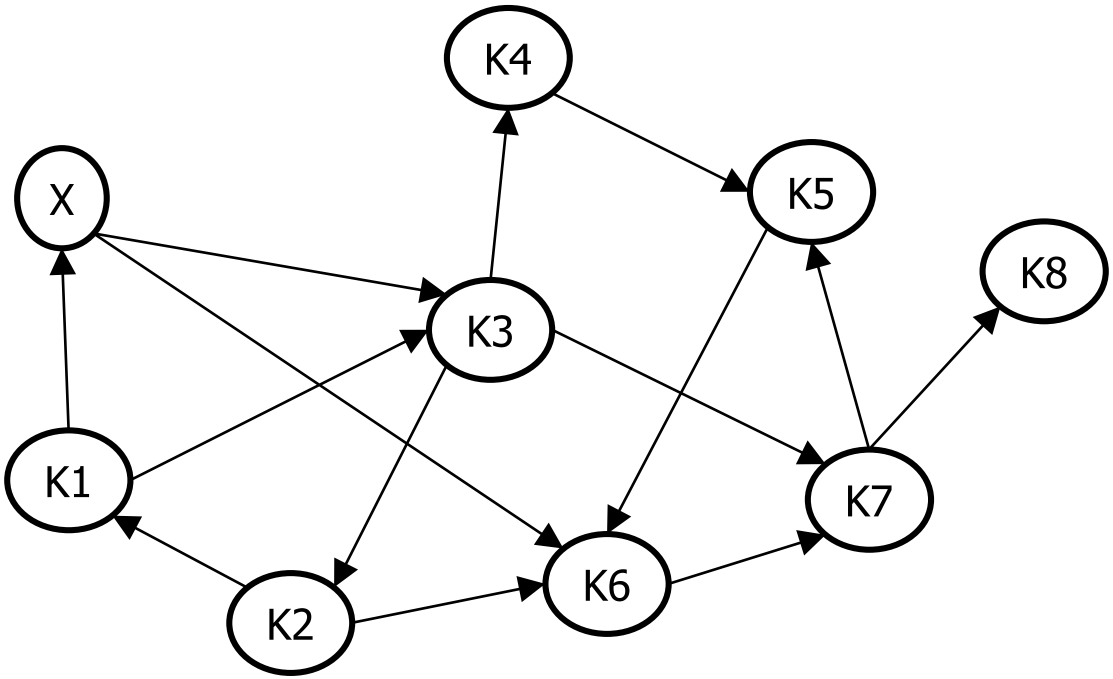
- Explanation
  - v can only be reached from w, if w is "higher" in the tree, i.e., v was traversed after w and hence has a higher preorder number
  - v can only be reached from w, if v is "lower" in the tree, i.e., v was left before w and hence has a lower postorder number
- Analysis: Test is O(1)
  - But preprocessing is O(n)
  - Pays off is pre-processed once, followed by many queries

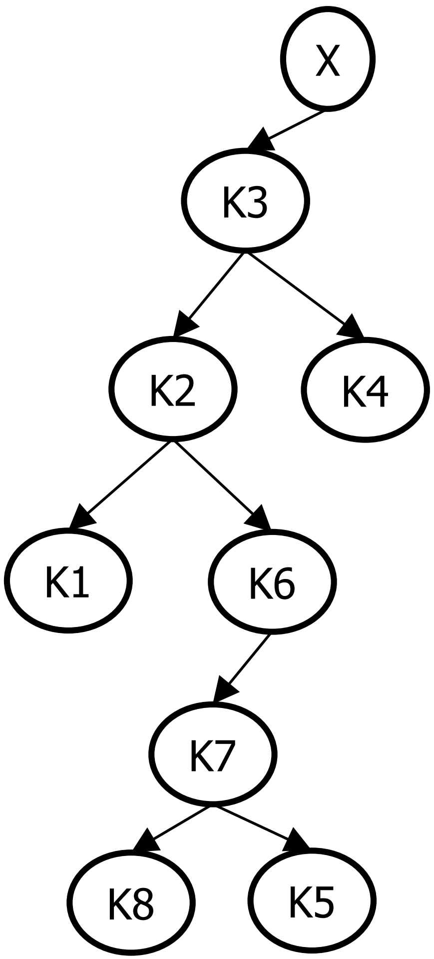


Pre-/Post-order Labeling for Graphs

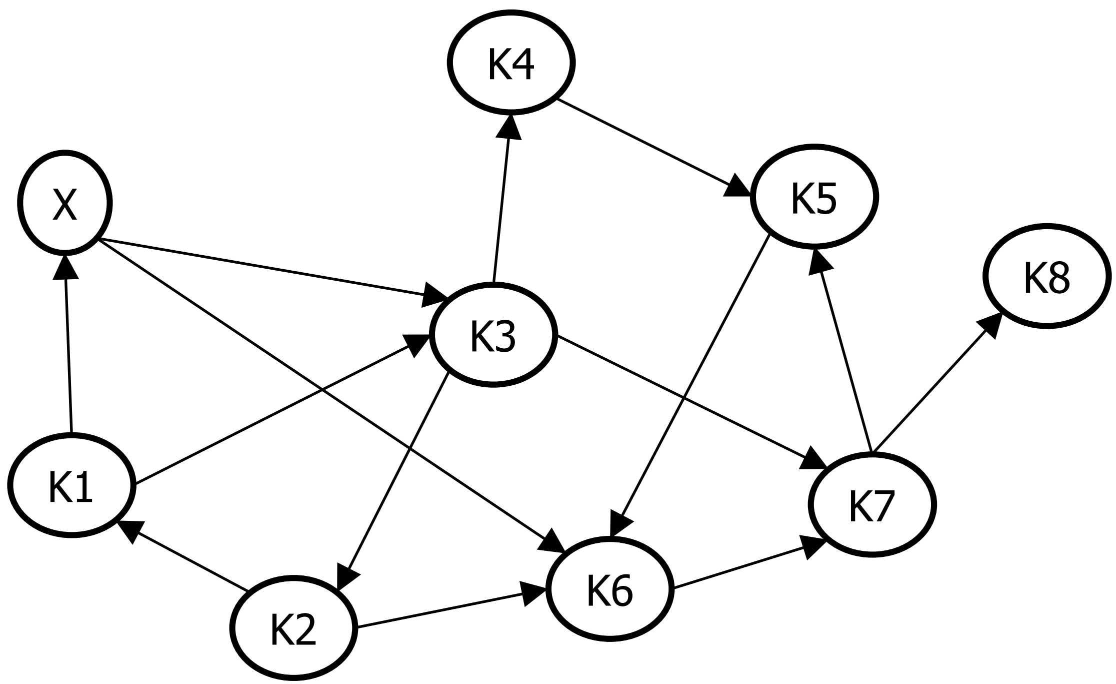
- Method
  Let G=(V, E). We assign each v∈V a pre-order and a post-order as follows. Set pre=post=1. Perform a depth-first traversal of G, starting at arbitrary nodes. When a node v is reached the first time, assign it the value of pre as pre-order value and increase pre. Whenever a node v is left the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

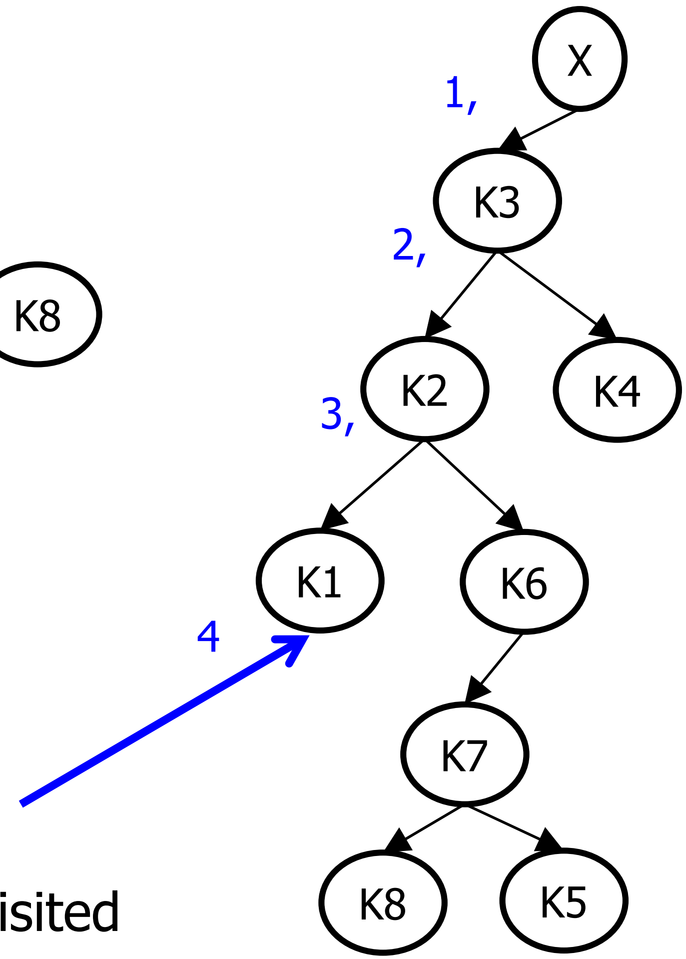
Example





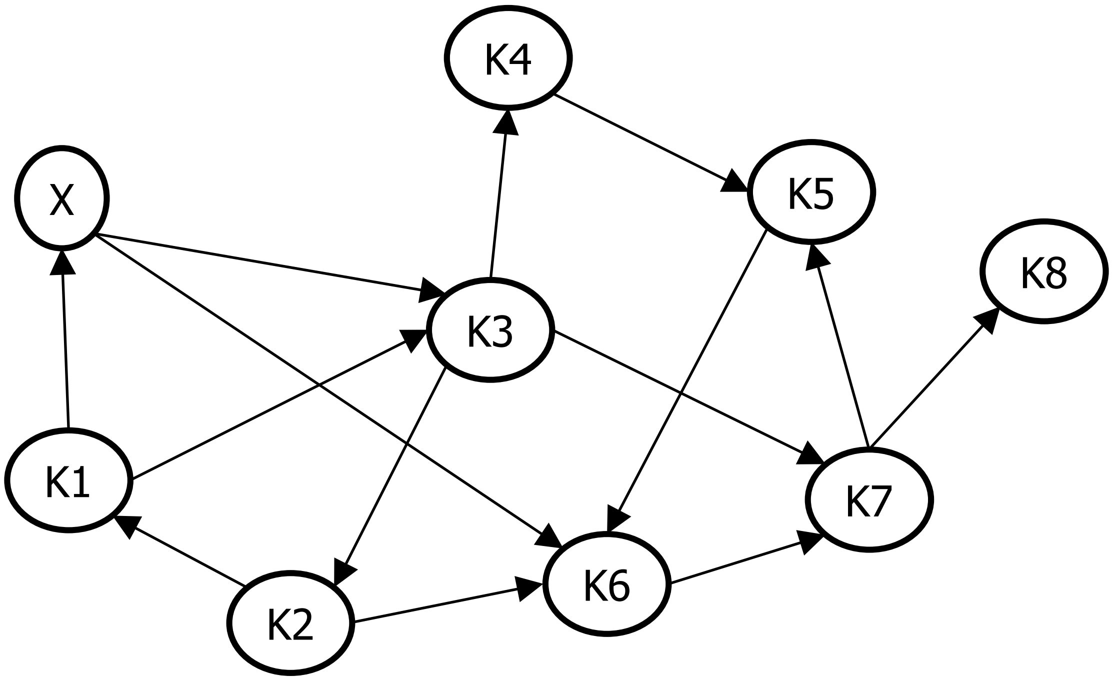
Example

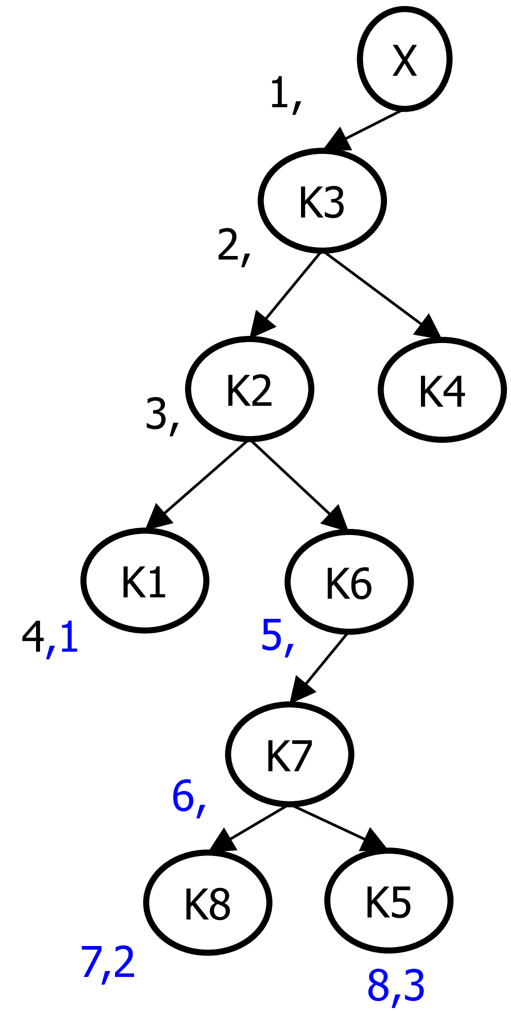




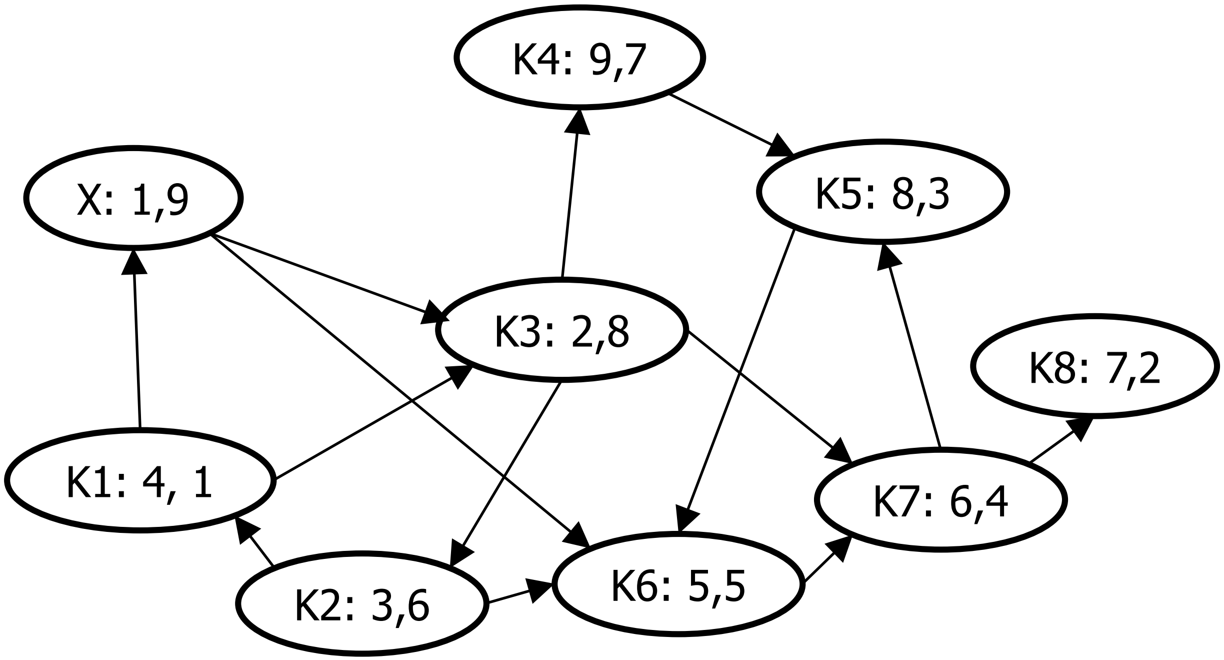
Last visit:
All children already visited

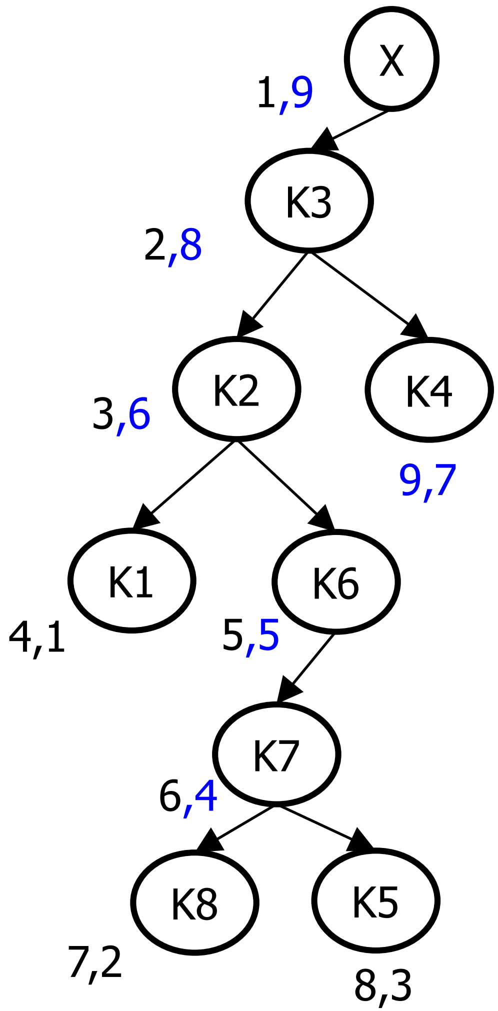
Example





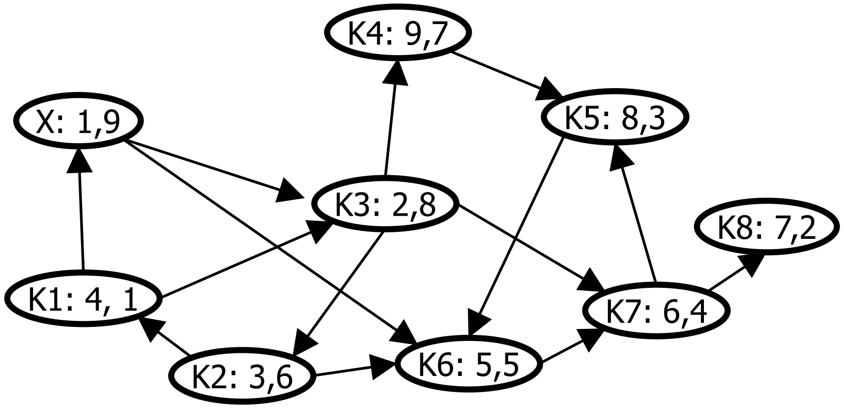
Example





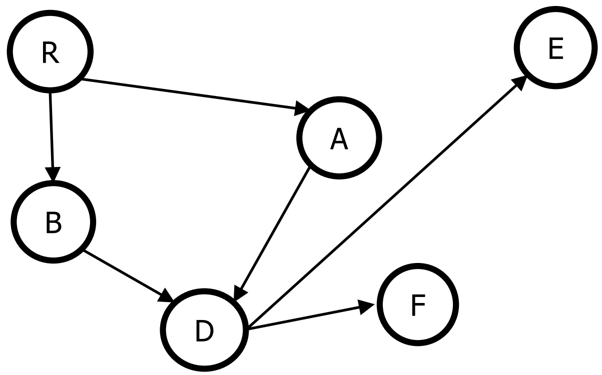
- Does our trick work?
  - Example: K1-K4
  - Reachable in G
  - But pre(K1)<pre(K4)
- Reachability trick does not work

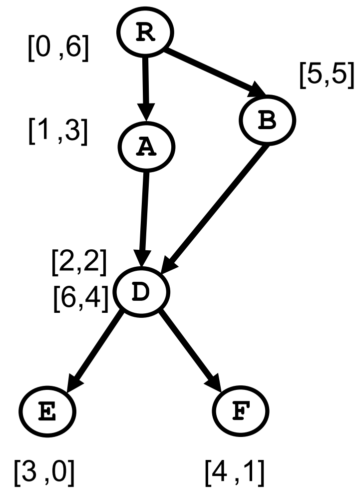
Ideas to Speed-Up Reachability in Graphs



- Much research over the last decade
  - PPO: Pre-/Post-Order Pair
- Trivial idea: Brute-Force
  - Assign to each node as many PPOs as paths that reach him
    - Choosing a set of roots is tricky
  - Reachability: Compare all pairs of PPOs of v and w (not O(1))
  - Requires exponential space in WC, depending on “tree-likeliness”
  - Efficient, if the graph is very “tree-like”
    - Single root, almost acyclic

Ideas to Speed-Up Reachability in Graphs: GRIPP

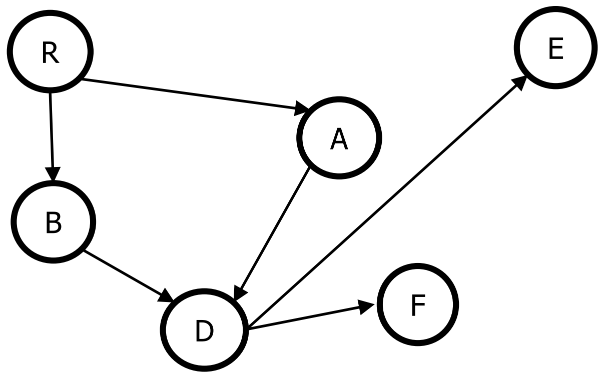


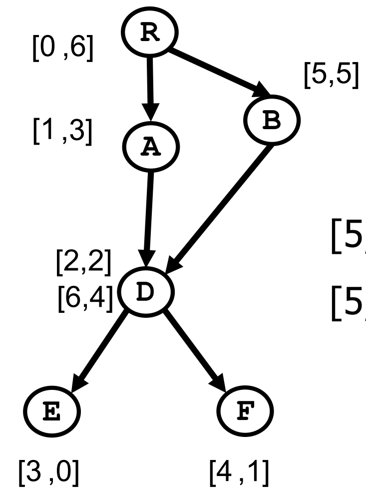


- GRIPP
  - If the graph is acyclic (wait)
  - Modified DFS: When a node is visited for the none-first time, assign another PPO but to not continue traversal further
  - During search, expand nodes in the PP-range of start nodes which have multiple PPOs
    - Expand: "Jump" to the all PPOs and branch 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.

Example





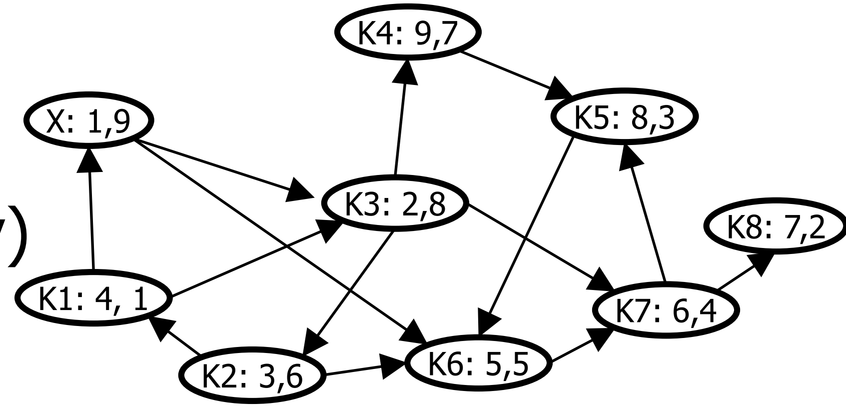
[5,5] – [3,0] = NO

[5,5] – [6,4]

[2,2] – [3,0] = YES

- Is E reachable from B?
  - First test: pre(E)<pre(B) – NO
  - But D is reachable from B (with second PPO)
  - Expand D – test its further PPOs
  - Second test (E reachable from D): YES

Tricks to Speed-Up Reachability: GRAIL



- 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 DFSs and use these 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 pairs are "reachable"
  - Parameter k controls runtime and space (trade-off)
  - Towards a probabilistic algorithm: Be very fast with high probability

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

Graph Indexing

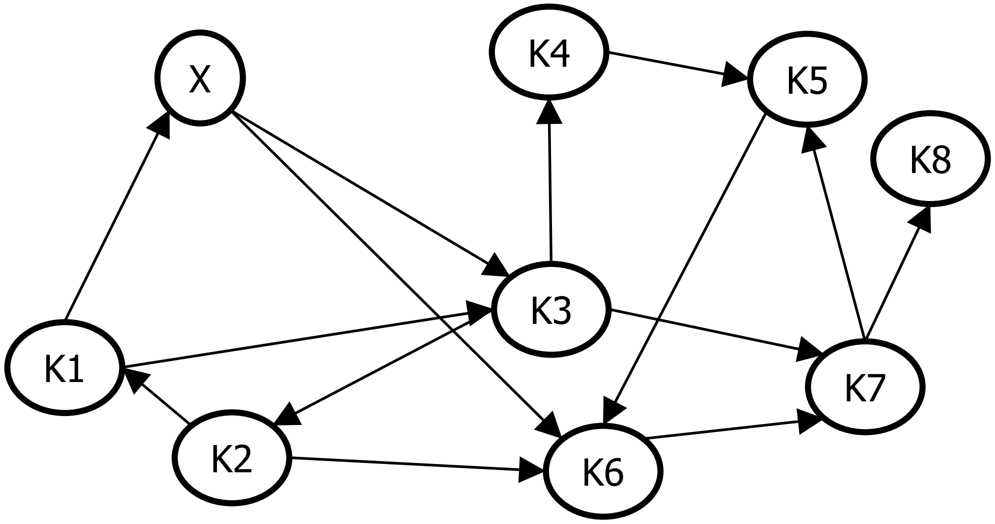
- Many other suggestions
  - Runtimes have been reduced since 2005 by at least a factor of 100
    - And graph sizes have grown by a factor of at least 1000
  - Current research: Timed graphs
    - Edges exist only in some windows in time (e.g.: ÖPNV)
    - Find a path and a start time when w is reachable from v
- All require a preprocessing phase (e.g. single or multiple 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
- Many 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

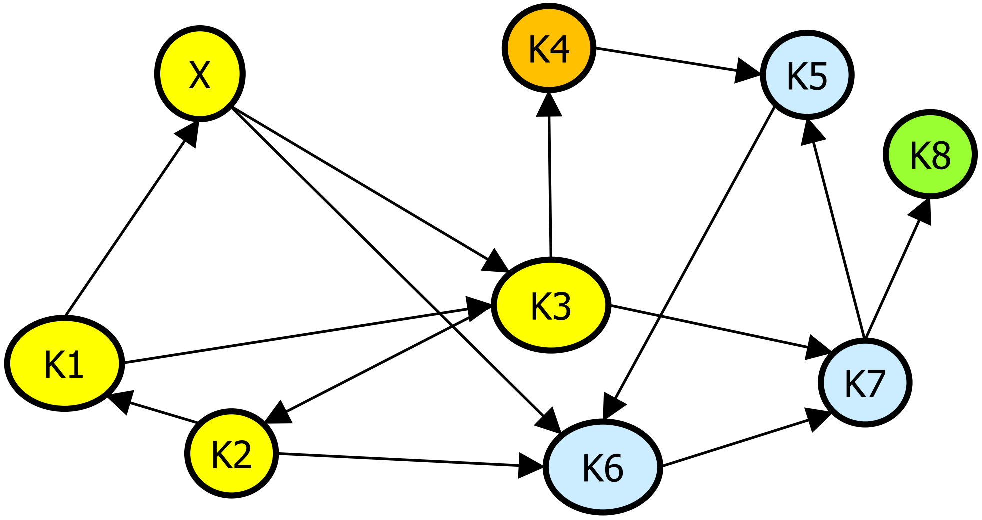
Recall: (Strongly) Connected Components

- 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*



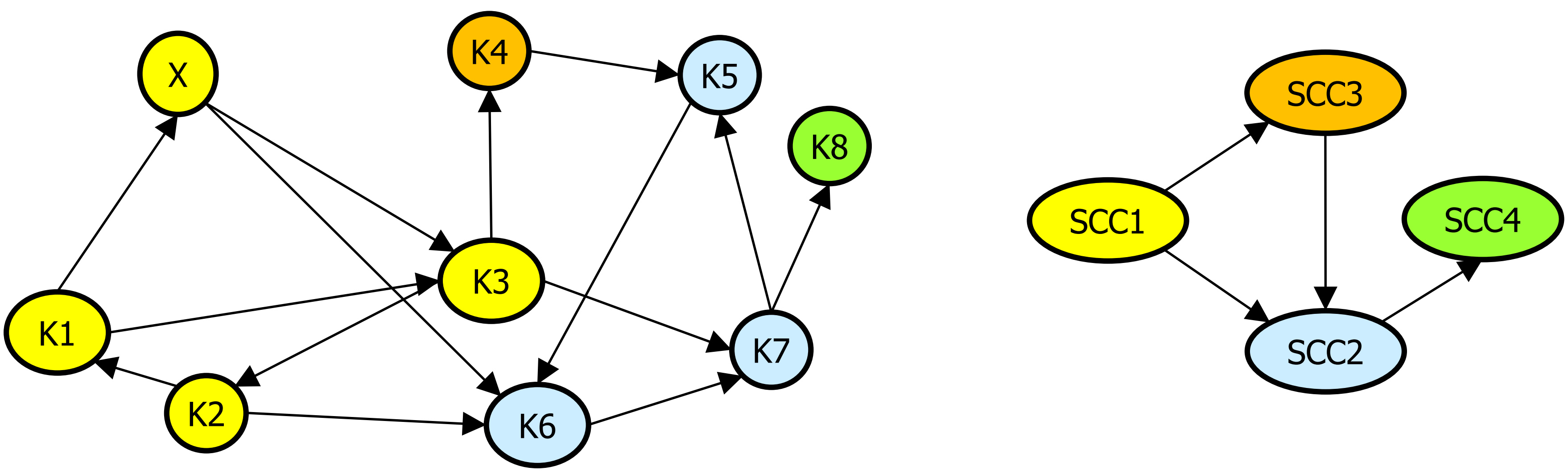
Recall

- 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)
  - Reachability v→w: If ssc(v)=ssc(w): yes; else: Look at G'
  - First path can be implemented in O(1) with hashing
- Computing SCC solves some problems in 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
- Question – how do we compute SCC(G)?

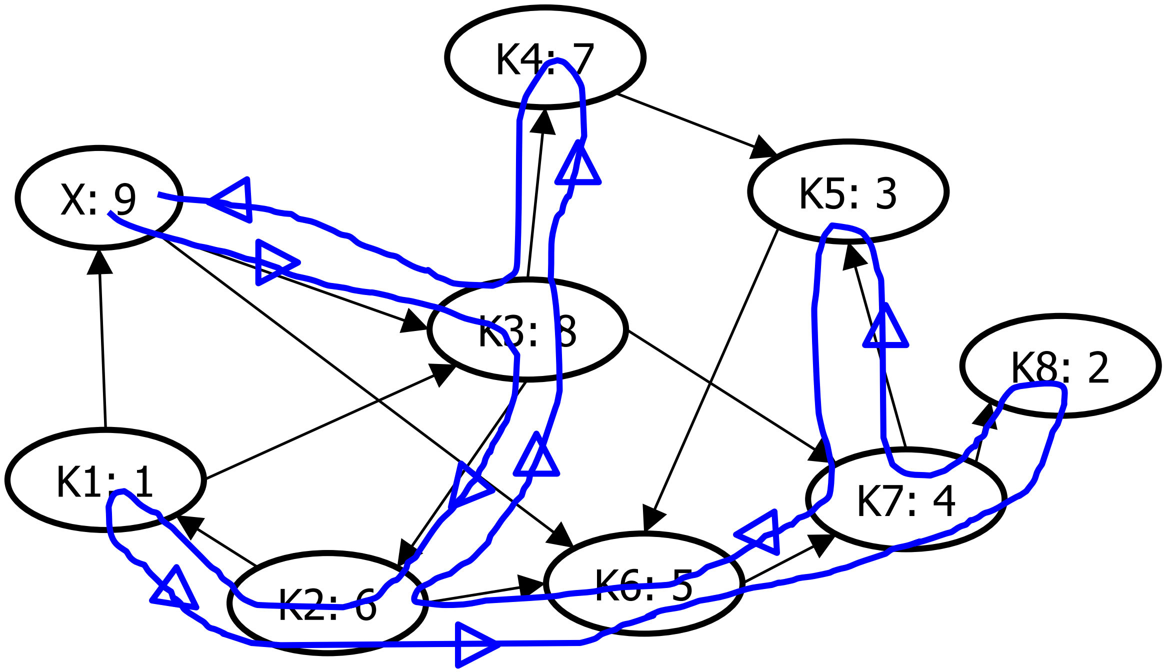
Content of this Lecture

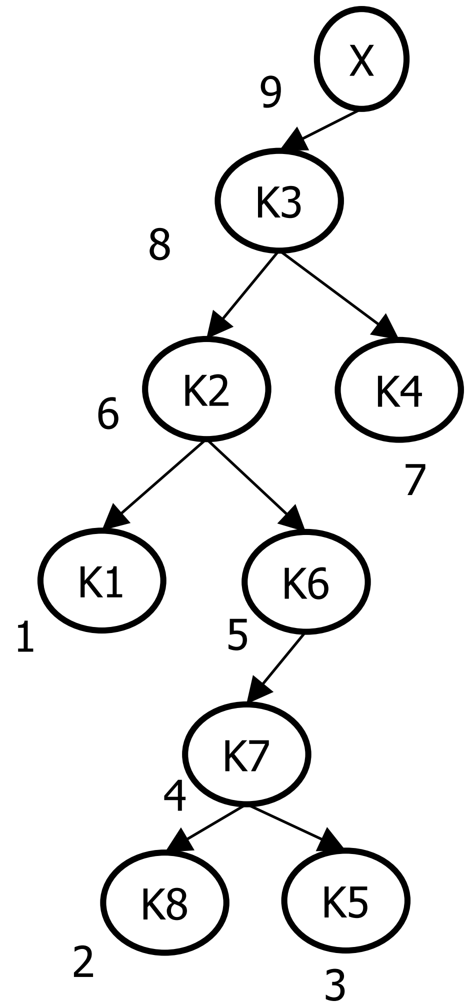
- Graph Traversals
- Strongly Connected Components (SCC)
  - Motivation
  - Kosaraju's algorithm

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
    - Break cycles; start as often until all nodes have a post-order
    - We don't need pre-order values
  - Compute G^T
  - Perform a second DFS on G^T always choosing as next root / node the one with the highest post-order according to the first DFS
  - All trees that emerge from the second DFS are SCC of G (and G^T)
- Kosaraju, 1978 (unpublished)

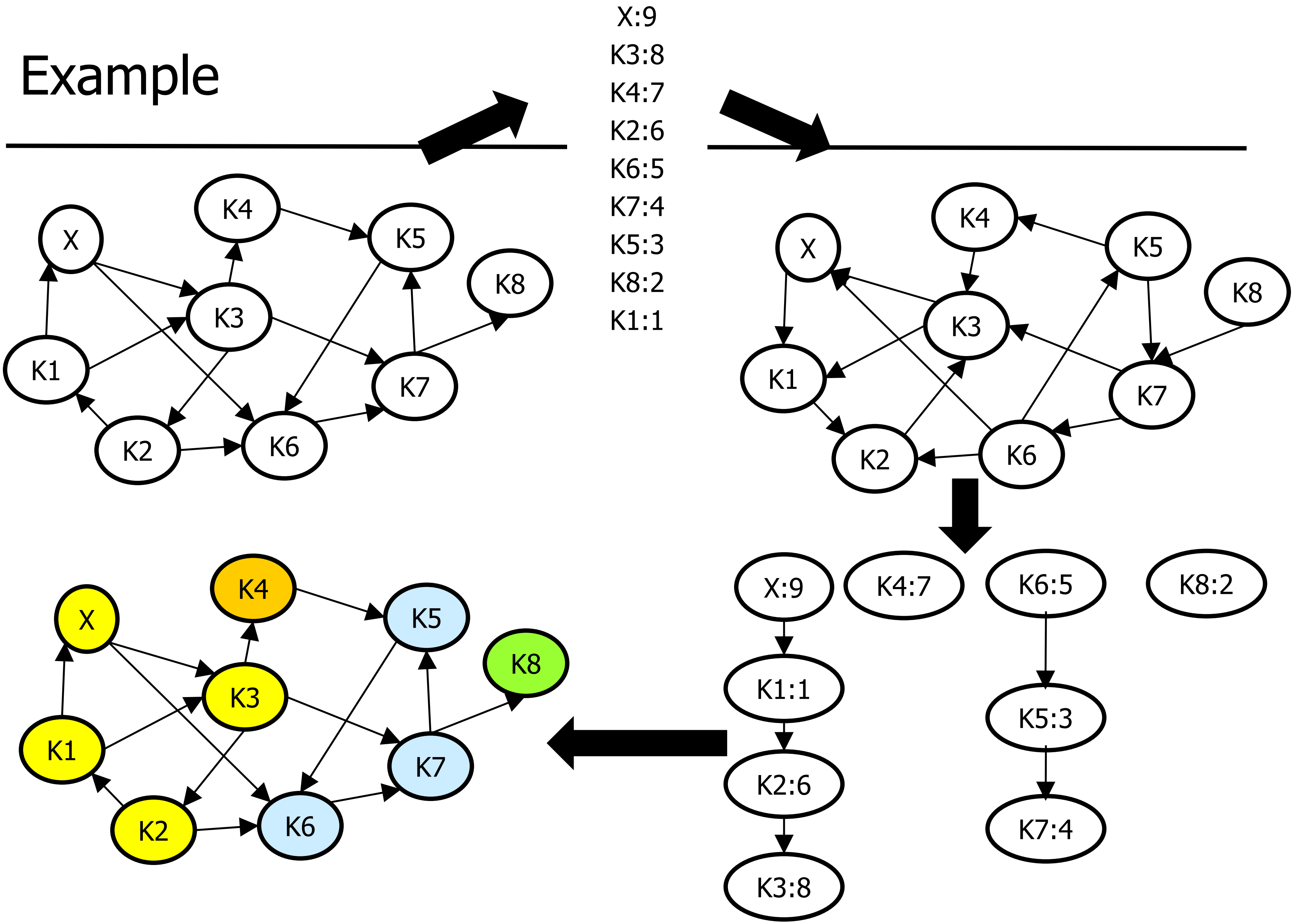
Example





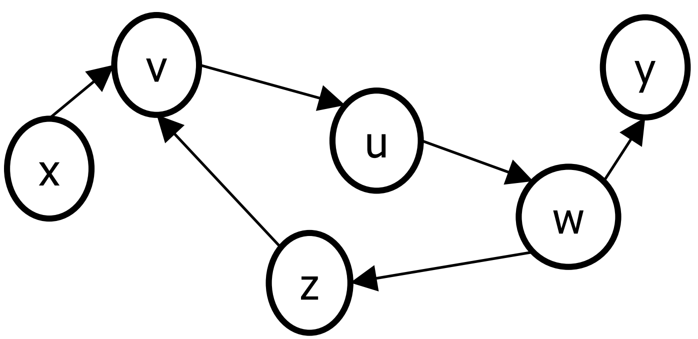
- Note: Usually, we need more than one root

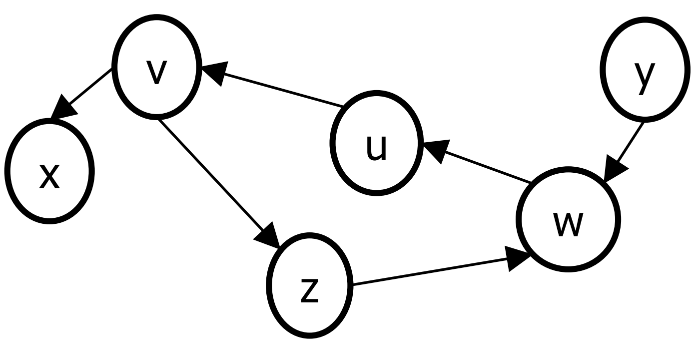
Example


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

Correctness

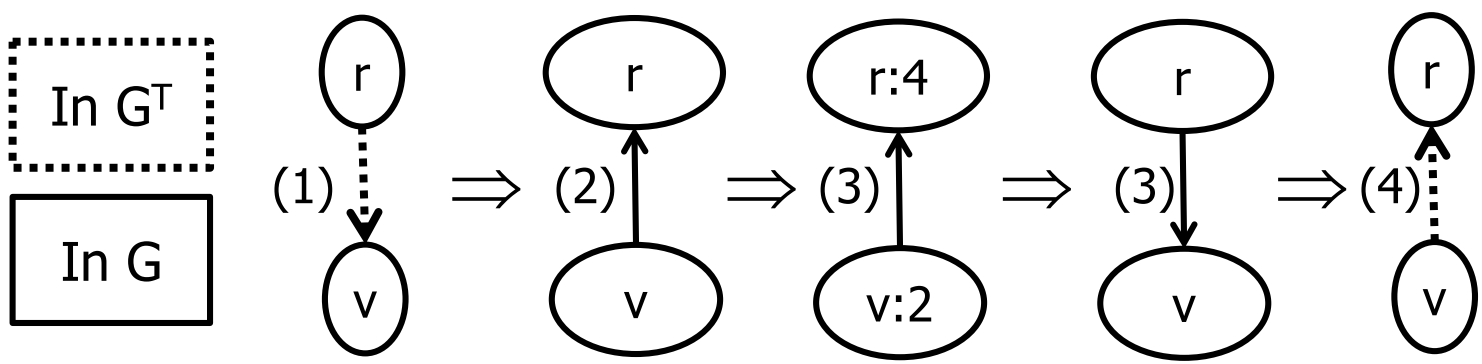
- Theorem
  Let G=(V,E). *Any two nodes v, w are in the same tree of the second DFS iff v and w are in the same SCC in G.*
- Proof
  - $\Leftarrow$: Suppose $v \rightarrow w$ and $w \rightarrow v$ in G. One of the two nodes (assume it is v) must be reached first during the second DFS. Since v can be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



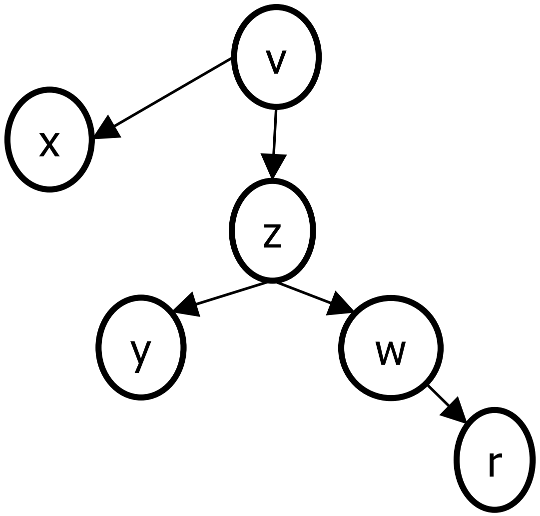


Correctness

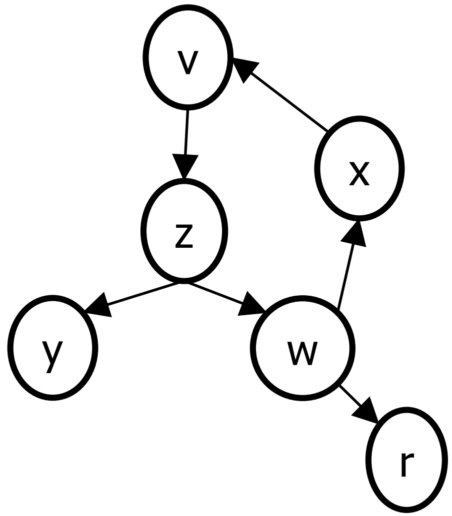
- $\Rightarrow$: Suppose v and w are in the same DFS-tree of G^T
  - Suppose r is the root of this tree
  - (1) Since $r \rightarrow v$ in G^T, it must hold that $v \rightarrow r$ in G
  - (2) Because of the order of the second DFS: post(r)>post(v) in G
  - (3) Thus, there must be a path $r \rightarrow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



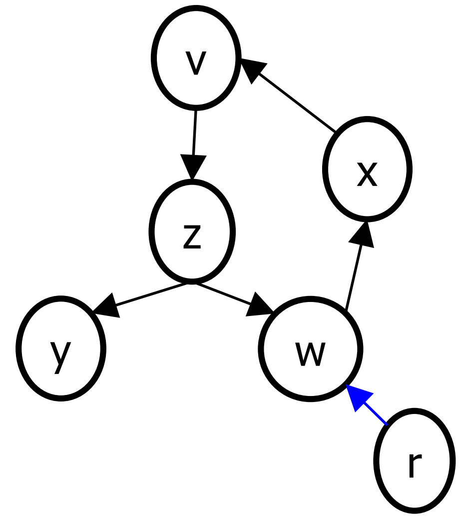
Examples (p(X) = post-order(X))



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



- $v \rightarrow w$ 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