Algorithms and Data Structures

Graphs 2: Shortest Paths (general case)
Strongly Connected Components

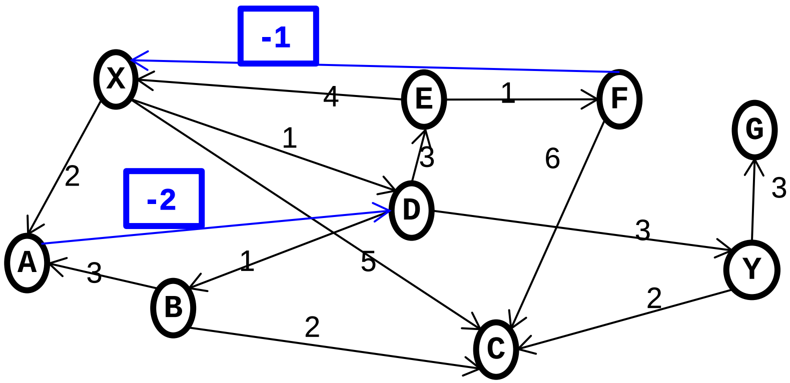
Ulf Leser

Content of this Lecture

- All-Pairs Shortest Paths
  - Transitive closure: Warshall's algorithm
  - Shortest paths: Floyd's algorithm
- Strongly Connected Components

All-Pairs Shortest Paths: General Case

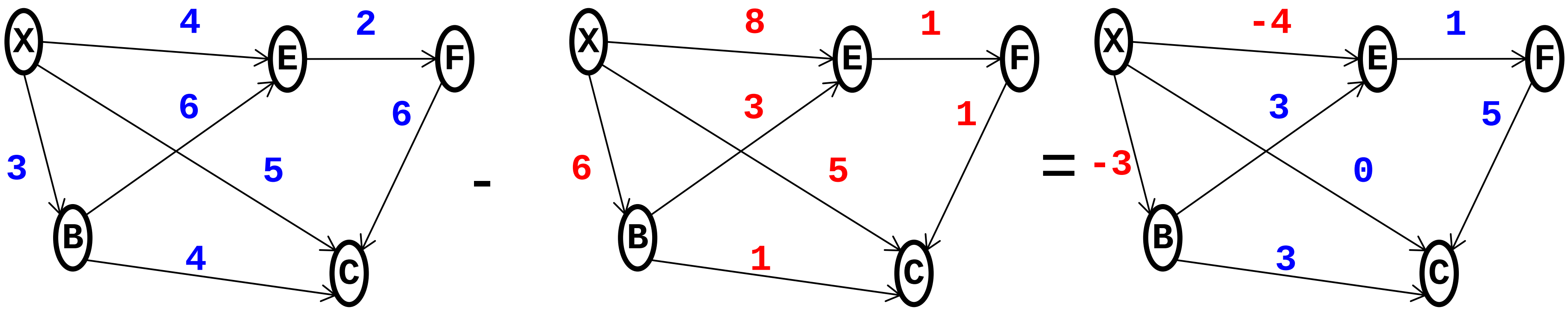
- All-pairs shortest paths: Given a digraph G with positive or negative edge weights, find the distance between all pairs of nodes
  - Transitive closure with distances
  - Result is $O(|V|^2)$ space, so don't try this for really large graphs



→	A	B	C	D	E	F	G	X	Y
A									
B									
C									
D									
E									
F									
G									
X									
Y									

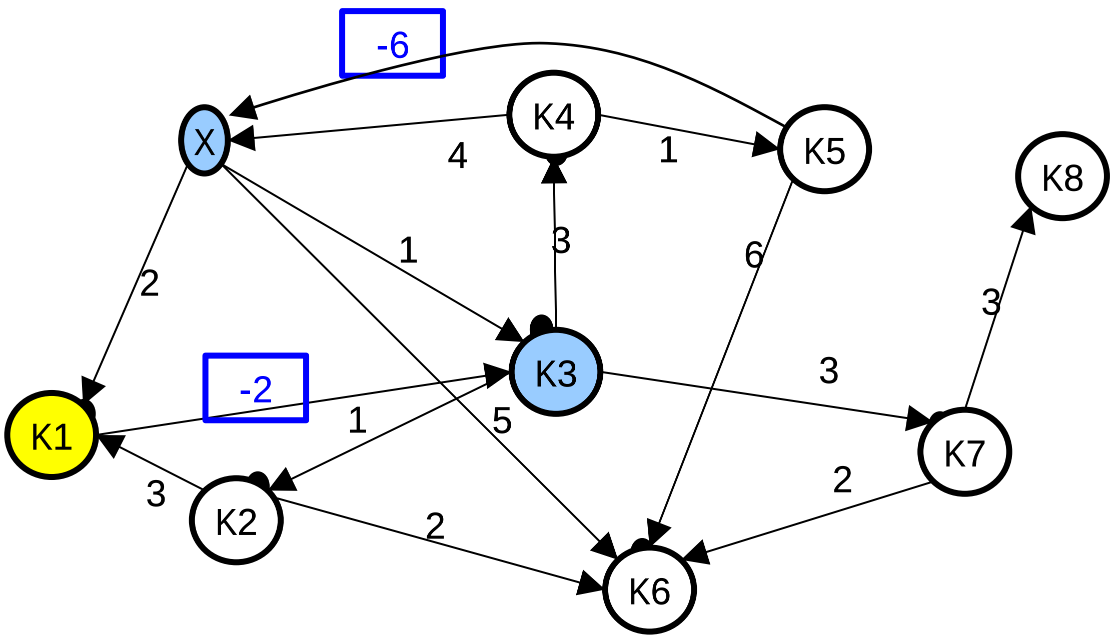
Why Negative Edge Weights?

- Transportation company
  - Every route incurs cost (for fuel, salary, etc.)
  - Every route creates income (for carrying the freight)
- If cost>income, edge weights become negative
  - But still important to find the best route
  - Example: Best tour from X to C



No Dijkstra

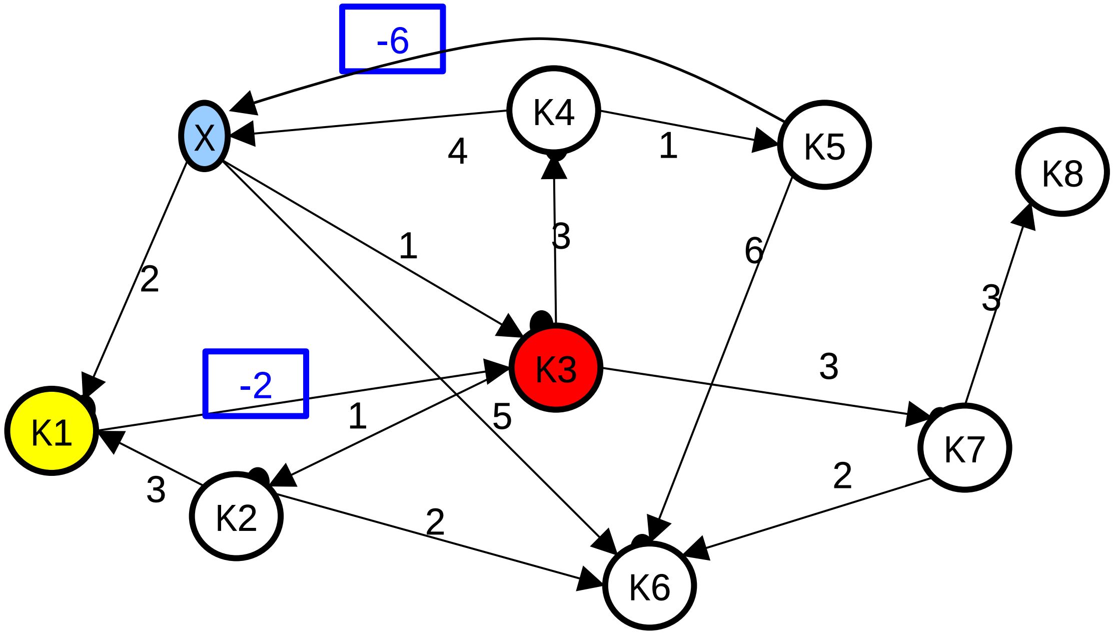
- Dijkstra does not work any more
  - Recall that Dijkstra enumerates nodes reachable by shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it “shorter” (by negative edge weights)



X	0
K1	2
K2	2
K3	1
K4	4
K5	
K6	5
K7	4
K8	

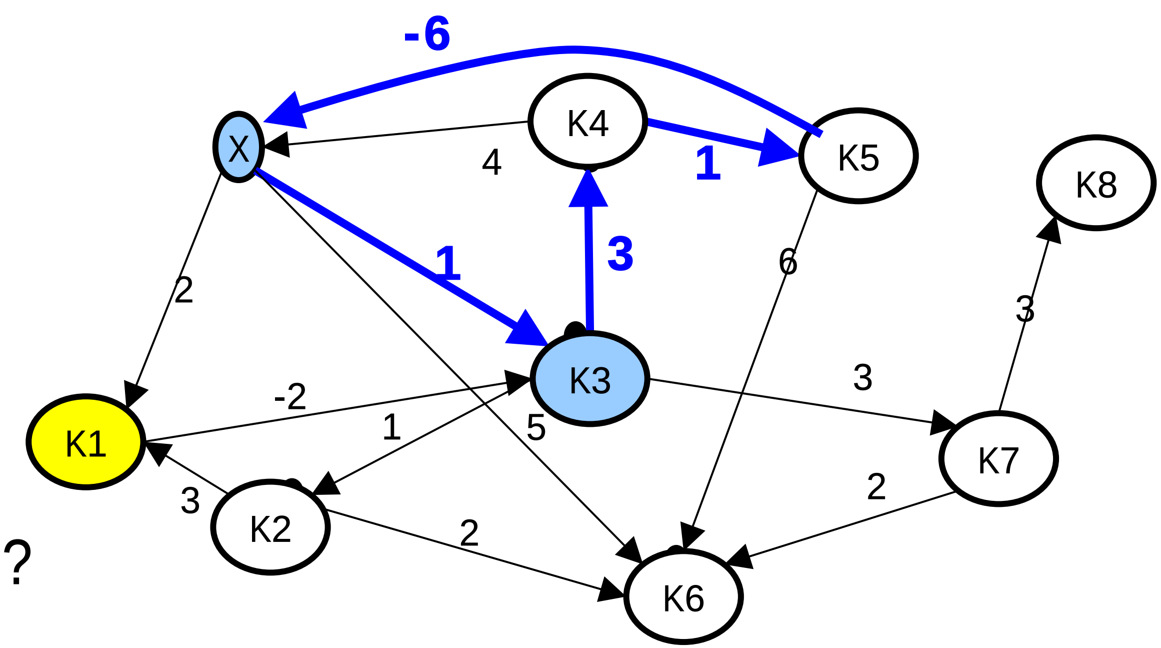
No Dijkstra

- Dijkstra does not work any more
  - Recall that Dijkstra enumerates nodes reachable by shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it “shorter” (by negative edge weights)



X	0
K1	0
K2	2
K3	0
K4	4
K5	
K6	5
K7	4
K8	

Negative Cycles



- Shortest path between X and K5?
  - X-K3-K4-K5: 5
  - X-K3-K4-K5-X-K3-K4-K5: 4
  - X-K3-K4-K5-X-K3-K4-K5-X-K3-K4-K5: 3
  - ...
- Problem is undefined if G contains a negative cycle

First Approach

- We start with a simpler problem: Computing the transitive closure of a digraph G without edge weights
- First idea
  - Reachability is transitive: $x \rightarrow y$ and $y \rightarrow z \Rightarrow x \rightarrow z$
  - We use this idea to iteratively build longer and longer paths
  - First extend edges with edges – path of length 2
  - Extend those paths with edges – paths of length 3
  - ...
  - No necessary path can be longer then |V|
- In each step, we store “reachable by a path of length ≤k” in a matrix

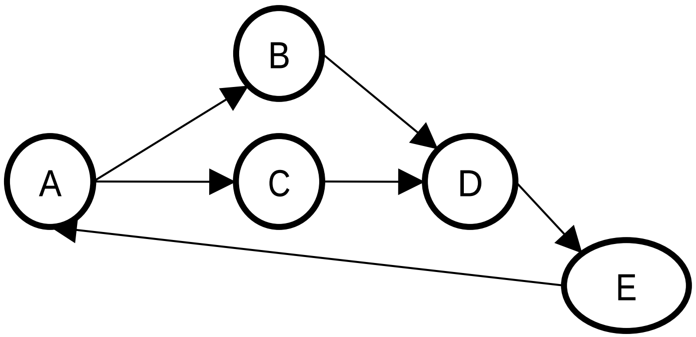
Naïve Algorithm

z appears nowhere ; it is just there to ensure that even longest shortest paths are found

```
G = (V, E);
M := adjacency_matrix( G);
M'' := M;
n := |V|;
for z := 1..n-1 do
  M' := M'';
  for i = 1..n do
    for j = 1..n do
      if M'[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M''[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
 end for;
end for;
```

- M is the adjacency matrix of G, M'' eventually the TC of G
- M': Represents paths ≤z
- Loops i and j look at all pairs reachable by a path of length at most z+1
- Loop k extends path of length at most z by all outgoing edges
- Analysis: Obviously $O(n^4)$

Example – After z=1, 2, 3, 4



	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A		1	1	1	1
B	1			1	1
C	1			1	1
D	1	1	1		1
E	1	1	1	1	

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

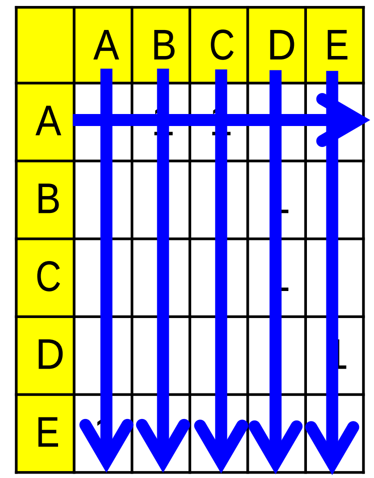
	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

Path length: ≤2 ≤3 ≤4 ≤5

Observation

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		



- In the first step, we actually compute M*M, and then replace each value ≥1 with 1
  - We only state that there is a path; not how many and not how long
- Transitive closure in graphs can be described as matrix operations

Paths in the Naïve Algorithm

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A		1	1	1	1
B	1			1	1
C	1			1	1
D	1	1	1		1
E	1	1	1	1	

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

- The naive algorithm always extends paths by one edge
  - I.e., it computes $M*M$, M^2*M, M^3*M, ... $M^{n-1}*M$

Idea for Improvement

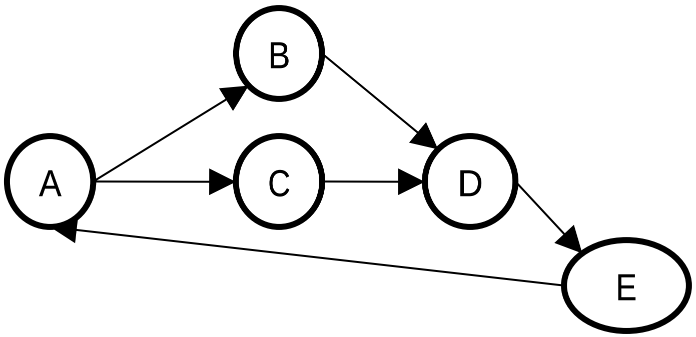
- Why not extend paths by all paths found so-far?
  - I.e., compute
    $M^{2'}=M*M$: Path of length at most 2
    $M^{3'}=M^{2'}*M \cup M^{2'}*M^{2'}$: Path of length 2+1 and 2+2
    $M^{4'}=M^{3'}*M \cup M^{3'}*M^{2'} \cup M^{3'}*M^{3'}$, Lengths 3/4+1, 3/4+1/2, 3/4+3/4
    ...
    $M^{n'}= \ldots \cup M^{n-1'}*M^{n-1'}$
  - [We will implement it differently]
- Trick: We can stop much earlier
  - The longest shortest path can be at most n
  - Thus, it suffices to compute $M^{\log(n)'}= \ldots \cup M^{\log(n)'}*M^{\log(n)'}$

Algorithm Improved

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for z := 0..ceil(log(n)) do
  for i = 1..n do
    for j = 1..n do
      if M[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
 end for;
end for;
```

- We use only one matrix M
- We "add" to M matrices M2', M3' ...
- In the extension, we see if a path of length $\leq 2^z$ (stored in M) can be extended by a path of length $\leq 2^z$ (stored in M)
  - Computes all paths $\leq 2*2^z=2^{z+1}$
- Analysis: $O(n^3*\log(n))$
- But ... we still can be faster

Example – After z=1, 2, 3



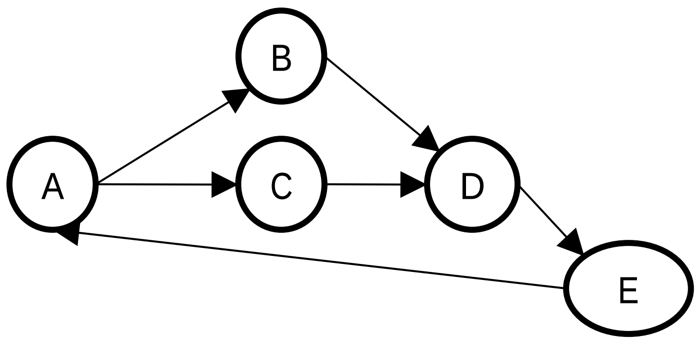
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

Path length: ≤2 ≤4 Done

Idea for Further Improvement



	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	A	B	C	D	E
A		1	1	1	
B				1	1
C				1	1
D	1				1
E	1	1	1		

- Note: The path A→D is found twice: A→B→D / A→C→D
- Can we stop searching A→D once we found A→B→D?
- Can we enumerate paths such that paths are discovered less often (i.e., less paths are tested)?

Warshall's Algorithm

- Warshall, S. (1962). A theorem on Boolean matrices. *Journal of the ACM 9(1): 11-12.*
- Key idea
  - Suppose a path $i \rightarrow k$ and $(i,k) \notin E$
  - Then there must be at least one node j with $i \rightarrow j$ and $j \rightarrow k$
  - Let j be the "smallest" such node (the one with the smallest ID)
  - If we fix the highest allowable ID t, then $i \rightarrow k$ is found iff $j \leq t$
  - Suppose we found all paths consisting only of nodes smaller than t
  - We increase t by one, i.e., we allow the usage of node t+1
  - Every new path must have the form $x \rightarrow (t+1) \rightarrow y$
- Enumerate paths by the IDs of the nodes they may use

Algorithm

- t gives the highest allowed node ID in a path
- Thus, node t must be on any new path
- We find all pairs i,k with i→t and t→k
- For every such pair, we set the path i→k to 1

```
G = (V, E);
M := adjacency_matrix(G);
n := |V|;
for t := 1..n do
  for i = 1..n do
    if M[i,t]=1 then
      for k=1 to n do
        if M[t,k]=1 then
          M[i,k] := 1;
        end if;
      end for;
    end if;
  end for;
end for;
```

Proof of Correctness

- Induction: Case t=1 is clear
- Going from t-1 to t
  - Induction assumption: We know all paths using only nodes with ID<t
  - We enter the i-loop
  - L6-L8 builds new paths over t
  - L6-L8 adds all paths which additionally contain the node with ID t
  - Induction assumption holds true
- These are all paths once t=n

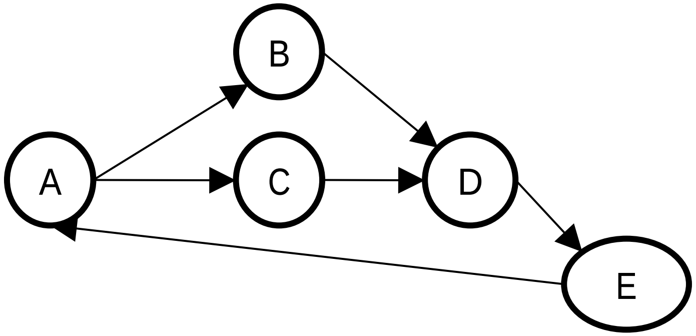
```
1.  G = (V, E);
2.  M := adjacency_matrix(
    G);
3.  n := |V|;
4.  for t := 1..n do
5.    for i = 1..n do
6.      if M[i,t]=1 then
7.        for k=1 to n do
8.          if M[t,k]=1 then
9.            M[i,k] := 1;
10.         end if;
11.       end for;
12.     end if;
13.   end for;
14. end for;
```

Example – Warshall's Algorithm

maxlen=2

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

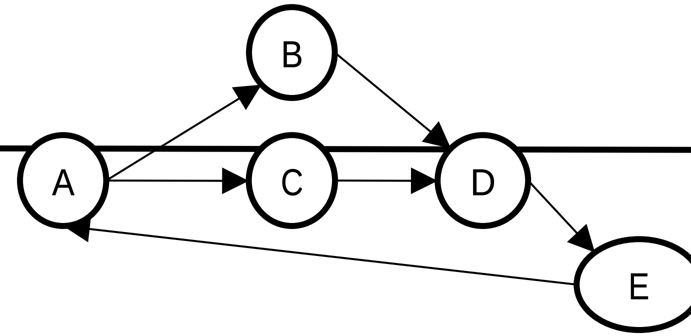
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1	1	1		



A allowed

Connect
E-A with
A-B, A-C

Example – After t=A,B,C,D,E



maxlen=2

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1	1	1		

=4

	A	B	C	D	E
A		1	1	1	
B				1	
C				1	
D					1
E	1	1	1	1	

=8

	A	B	C	D	E
A		1	1	1	
B				1	
C				1	
D					1
E	1	1	1	1	

	A	B	C	D	E
A		1	1	1	1
B				1	1
C				1	1
D					1
E	1	1	1	1	1

	A	B	C	D	E
A	1	1	1	1	1
B	1	1	1	1	1
C	1	1	1	1	1
D	1	1	1	1	1
E	1	1	1	1	1

B allowed

Connect A-B/E-B with B-D

C allowed

Connect A-C/E-C with C-D

No news

D allowed

Connect A-D, B-D, C-D with D-E

E allowed

Connect everything with everything

Little change – Dramatic Consequences

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for z := 1..n do
  for i = 1..n do
    for j = 1..n do
      if M[i,j]=1 then
        for k=1 to n do
          if M[j,k]=1 then
            M[i,k] := 1;
          end if;
        end for;
      end if;
    end for;
  end for;
end for;
```

Swap i and j loop

Rephrase j into t

```
G = (V, E);
M := adjacency_matrix( G);
n := |V|;
for t := 1..n do
  for i = 1..n do
    if M[i,t]=1 then
      for k=1 to n do
        if M[t,k]=1 then
          M[i,k] := 1;
        end if;
      end for;
    end if;
  end for;
end for;
```

$O(n^4)$

$O(n^3)$

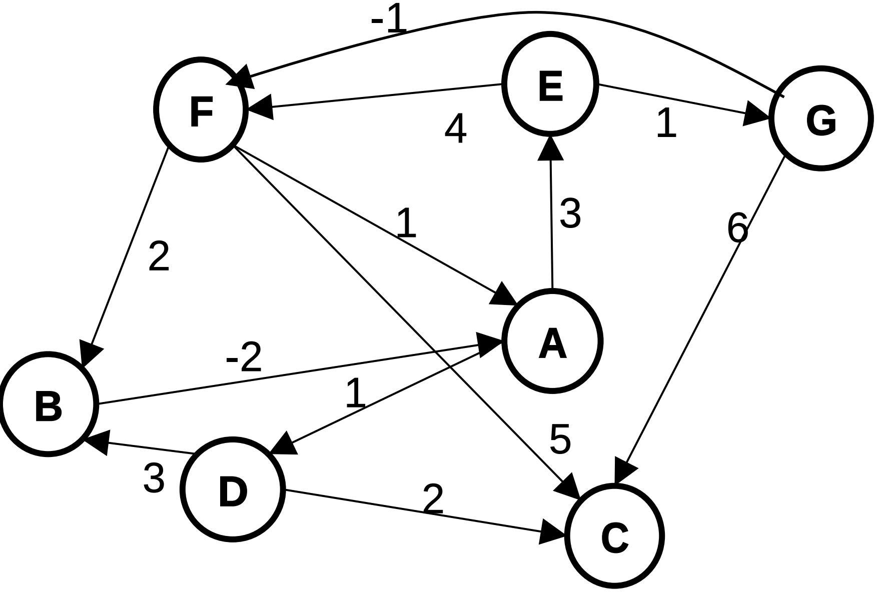
Content of this Lecture

- All-Pairs Shortest Paths
  - Transitive closure: Warshall's algorithm
  - Shortest paths: Floyd's algorithm
- Strongly Connected Components

Shortest Paths

- Floyd, R. W. (1963). Algorithm 97: Shortest Path. *Communications of the ACM 5(6): 345.*
- We use the same idea: Enumerate paths using only nodes smaller than t
- Invariant: Before step t, M[i,j] contains the length of the shortest path that uses no node with ID higher than t
- When increasing t, we find new paths $i \rightarrow t \rightarrow k$ and look at their lengths
- Thus: $M[i,k] := \min(M[i,k] \cup \{ M[i,t]+M[t,k] \mid i \rightarrow t \wedge t \rightarrow k\})$

Example



	A	B	C	D	E	F	G
A				1	3		
B	-2						
C							
D		3	2				
E						4	1
F	1	2	5				
G			6			-1	

	A	B	C	D	E	F	G
A				1	3		
B	-2			-1	1		
C							
D		3	2				
E						4	1
F	1	2	5	2	4		
G			6			-1	

	A	B	C	D	E	F	G
A				1	3		
B	-2			-1	1		
C							
D	1	3	2	2	4		
E						4	1
F	0	2	5	1	3		
G			6			-1	

Summary

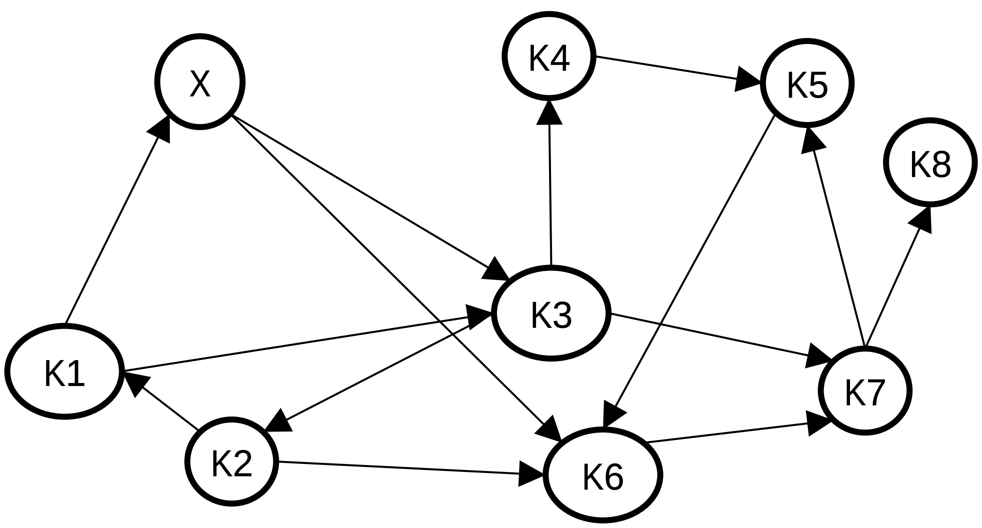
- Warshall‘s algorithm computes the transitive closure of any digraph G without negative cycles in $O(|V|^3)$
- Floyd‘s algorithm computes the distances between any pair of nodes in a digraph without negative cycles in $O(|V|^3)$
- Storing both information requires $O(|V|^2)$
- Problem is easier for ...
  - undirected graphs: Connected components
  - graphs with only positive edge weights: All-pairs Dijkstra
  - trees: See nice problems

Content of this Lecture

- All-Pairs Shortest Paths
- Strongly Connected Components (SCC)
  - Why?
  - Pre/Postorder Traversal
  - Kosaraju's algorithm

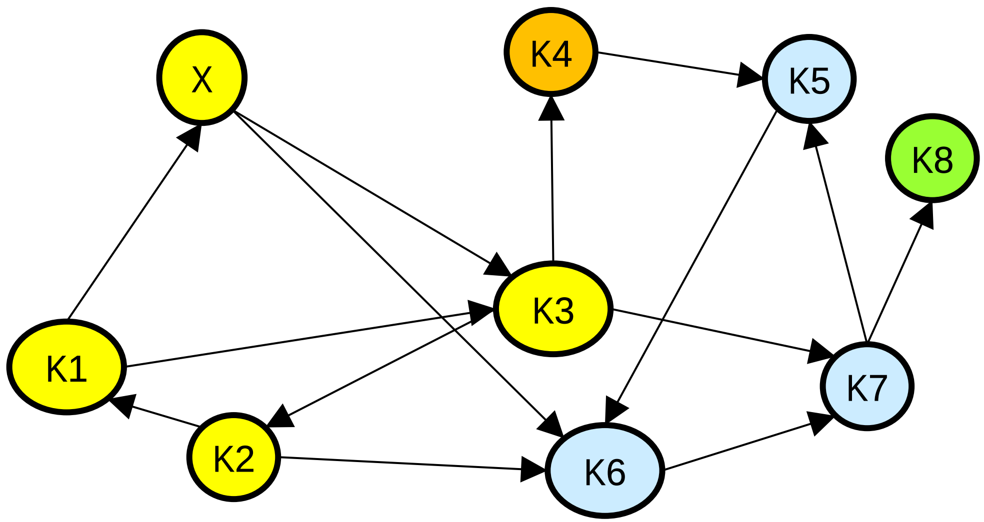
Recall

- Definition
  Let G=(V, E) be a directed graph.
  - *A 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*



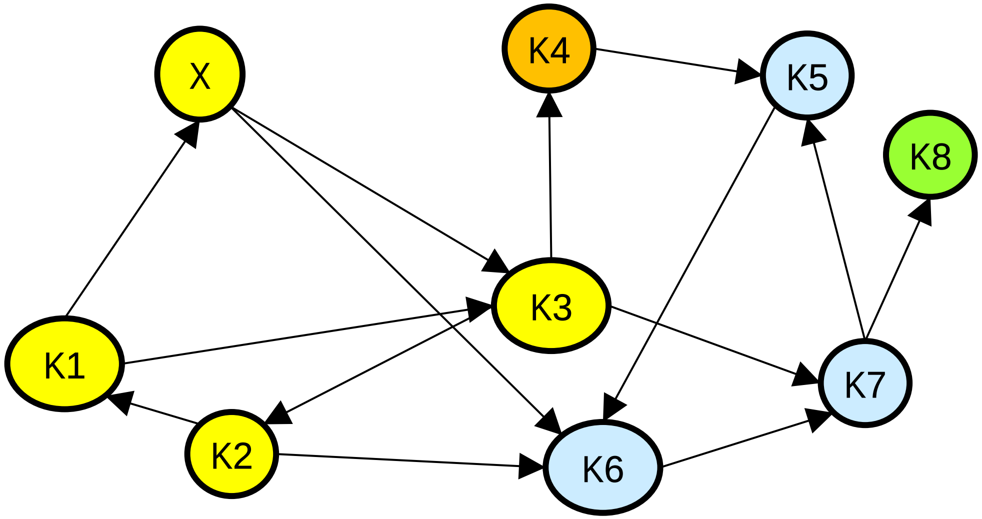
Recall

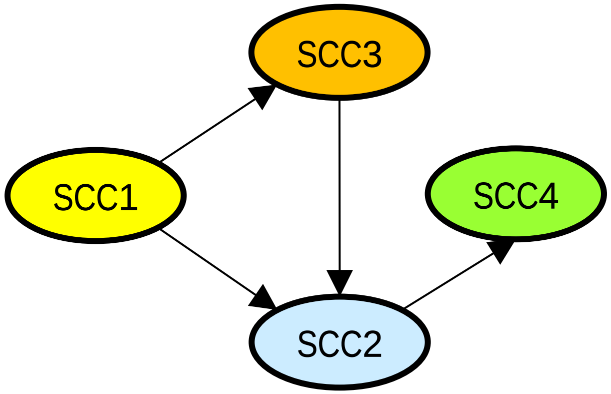
- Definition
  Let G=(V, E) be a directed graph.
  - *A subgraph G'=(V', E') of G is called connected if G' contains a path between any pair v,v'∈V'*
  - *Any maximal connected subgraph of G is called a strongly connected component of G*



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)

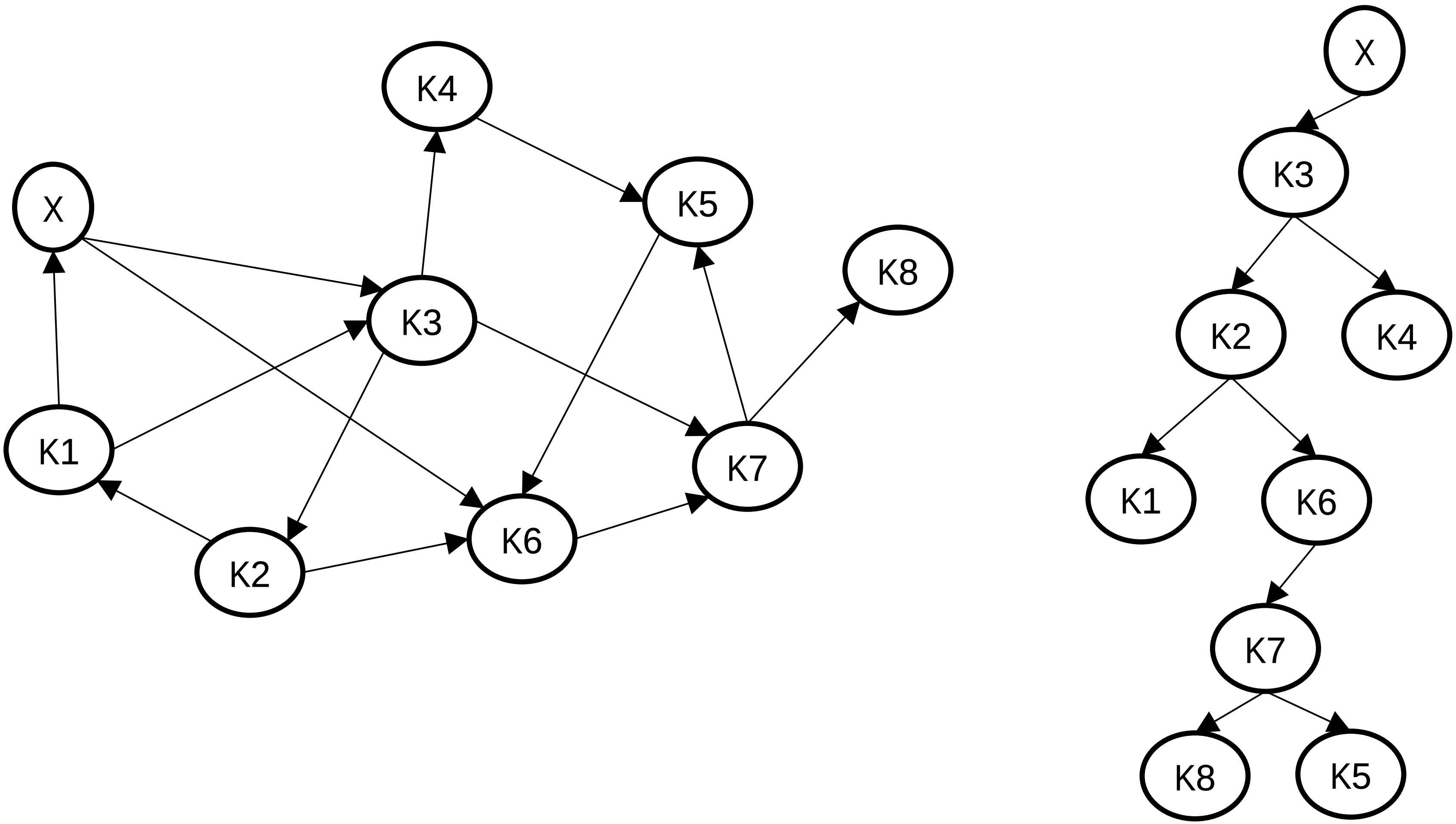




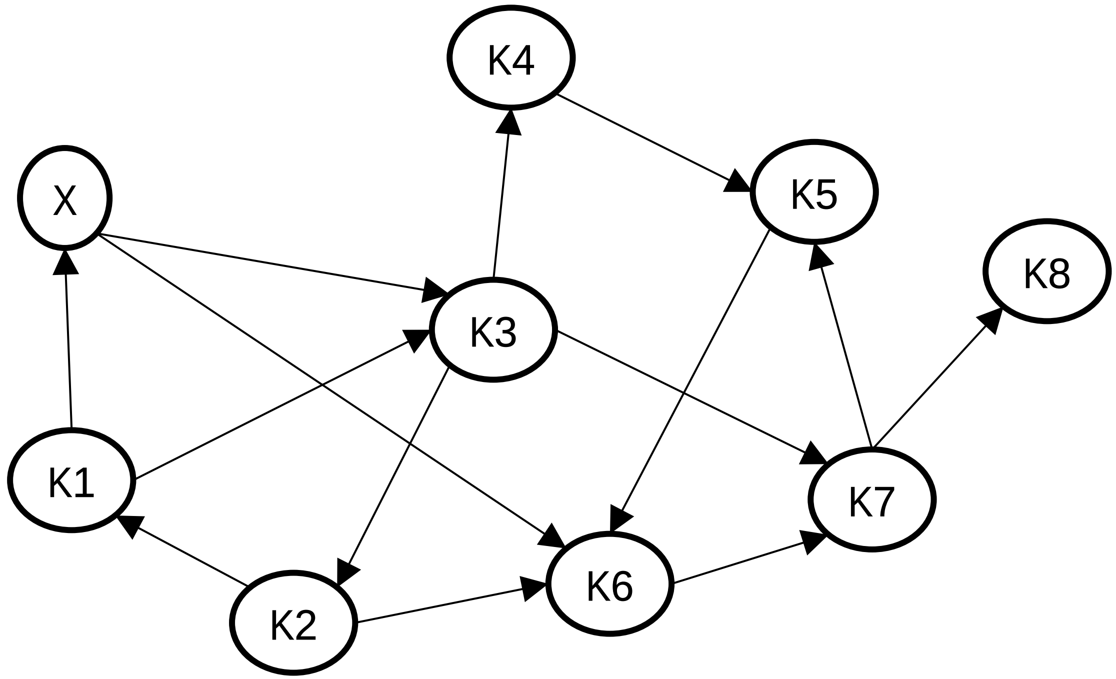
Graph Traversal

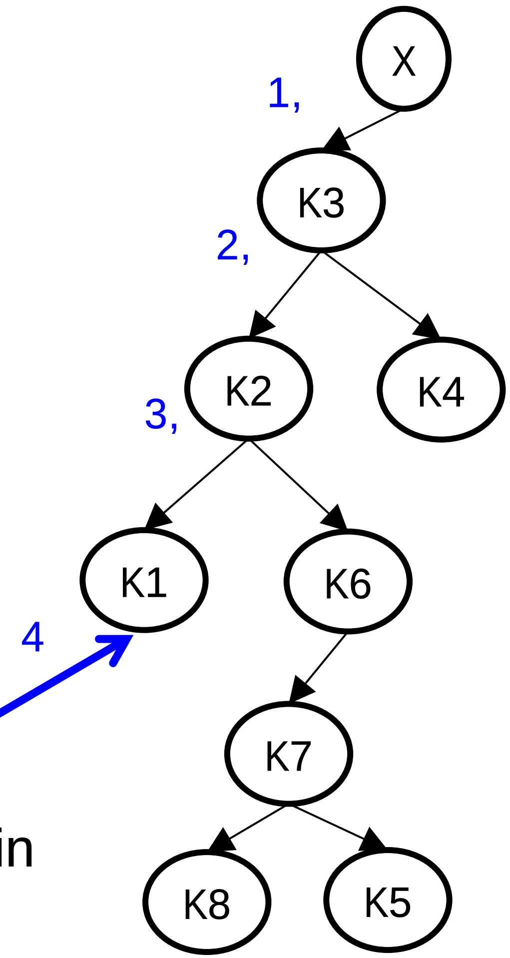
- Most algorithms for finding SCC are based on pre-/post-order labeling of nodes
- Method
  Let G=(V, E). We assign each v∈V a pre-order and a post-order in the following way. Set counters pre=post=0. Perform a depth-first traversal of G. Whenever a node v is reached the first time, assign it the value of pre as pre-order value and increase pre. Whenever a node v is left the last time, assign it the value of post as post-order value and increase post.
- Obviously O(|G|)
  - Labeling not unique; depends on order in which children are visited

Example



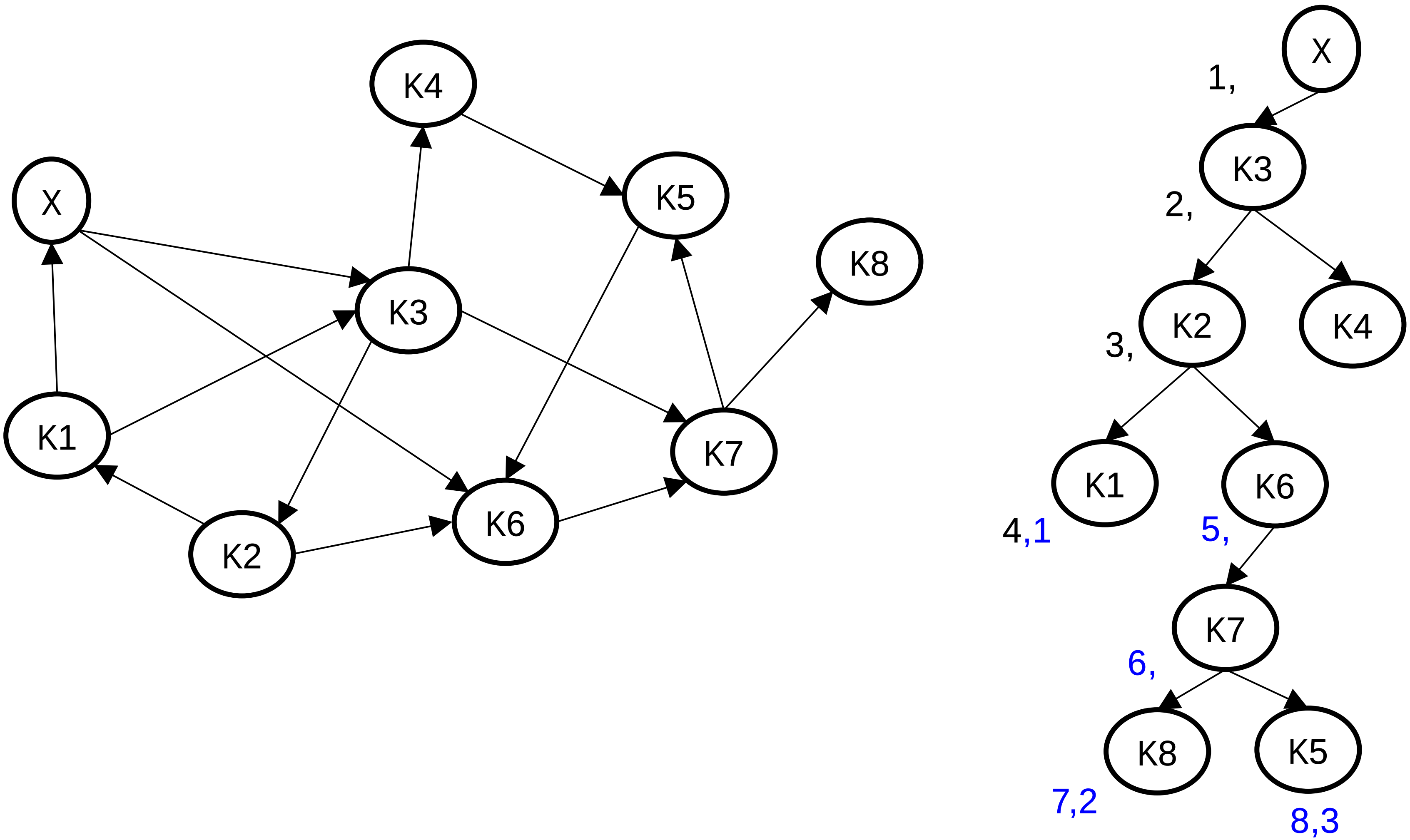
Example



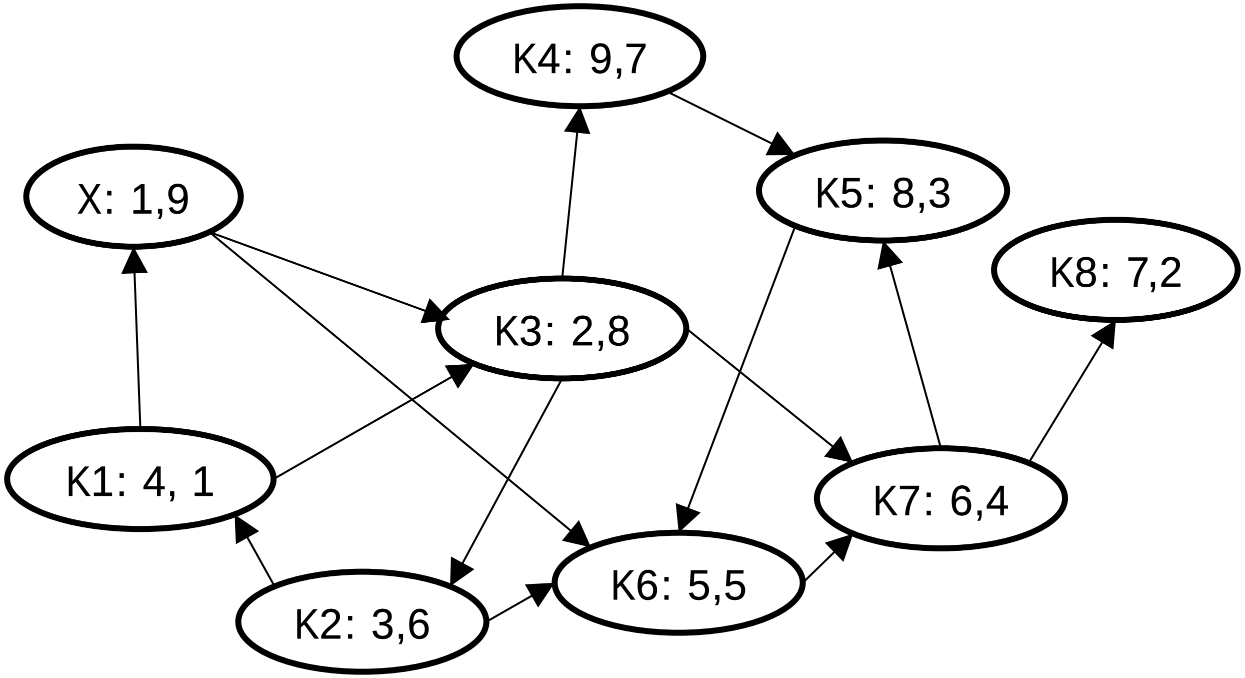


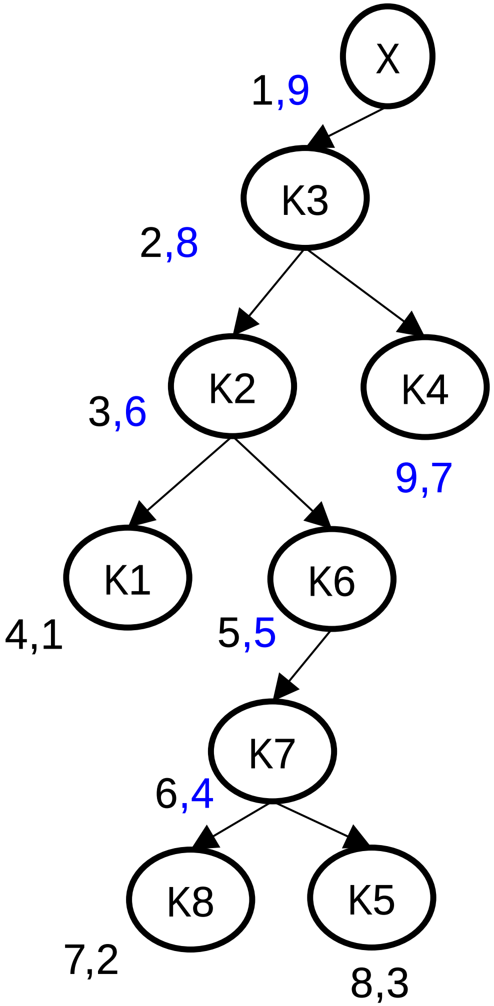
Last visit: Cannot be visited again
without running into a cycle

Example



Example


K4: 9,7
X: 1,9
K5: 8,3
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

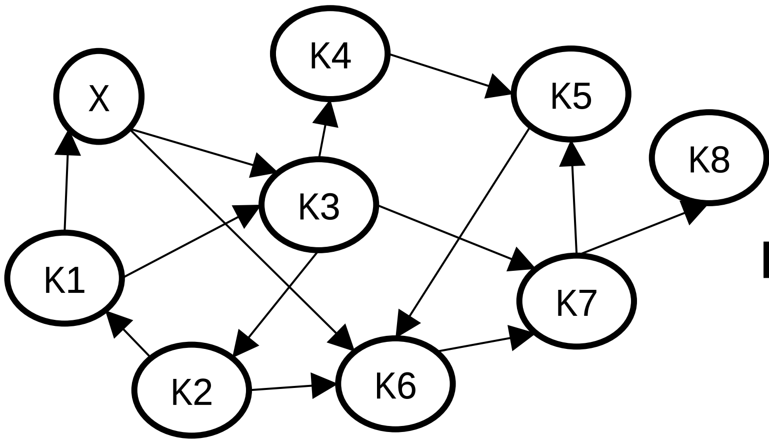
Content of this Lecture

- All-Pairs Shortest Paths
- Strongly Connected Components (SCC)
  - Why?
  - Pre/Postorder Traversal
  - Kosaraju's algorithm

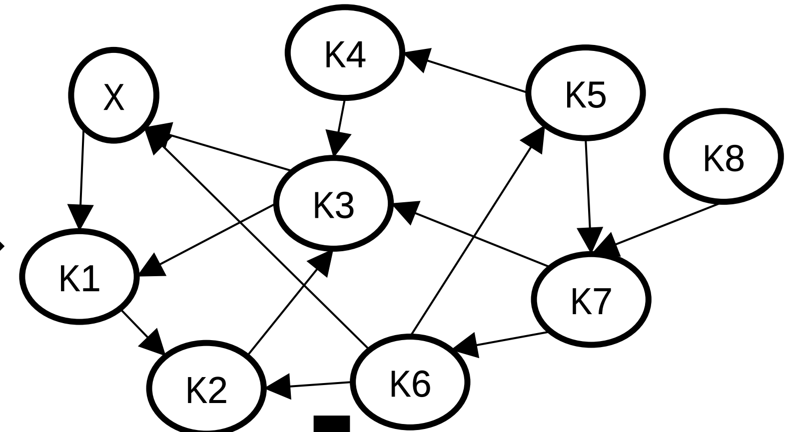
Kosaraju's Algorithm

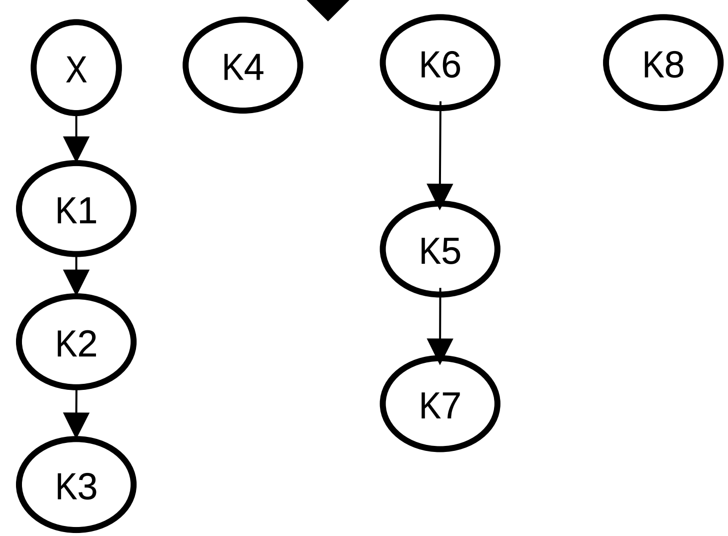
- Definition
  Let $G=(V,E)$. The graph $G^T=(V, E')$ with $(v,w) \in E'$ iff $(w,v) \in E$ is called the transposed graph of G.
- Kosaraju's algorithm is very short
  - Compute post-order labels for all nodes from G using a first DFS
    - Here, we actually don't need the pre-order values
  - Compute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

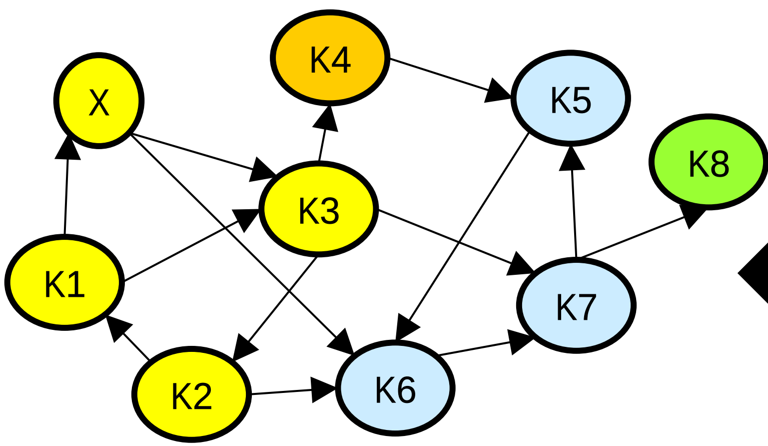
Example



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

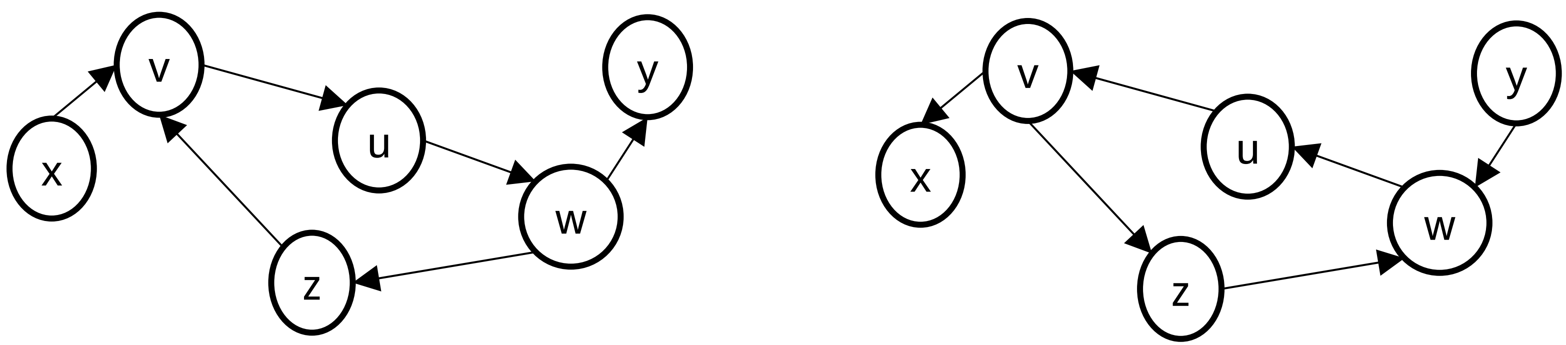






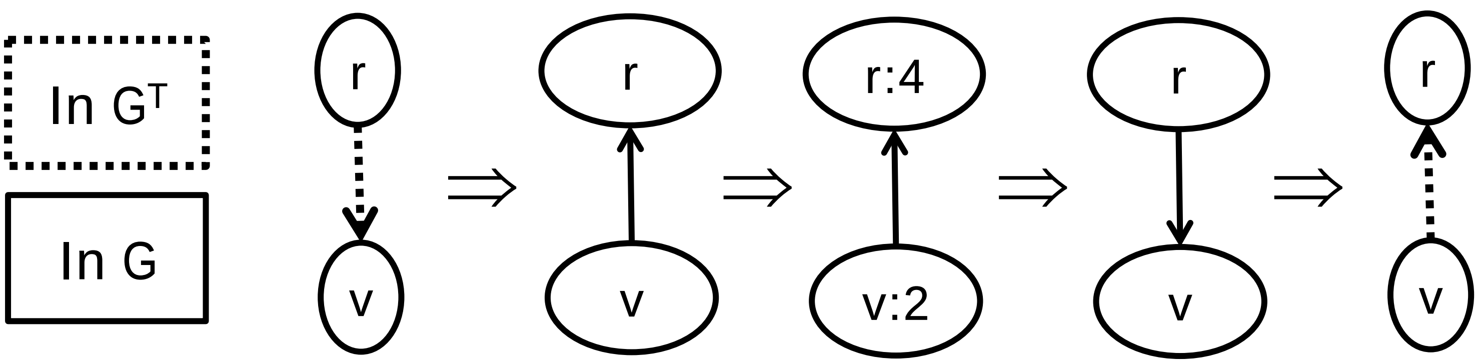
Correctness

- 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 in the same tree, so they are in the same tree of G^T.

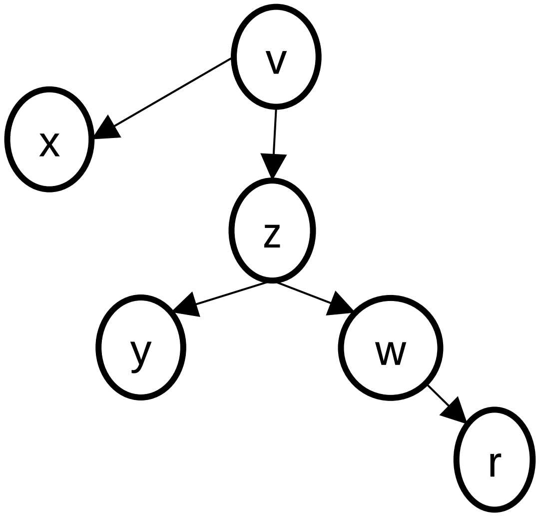


Correctness

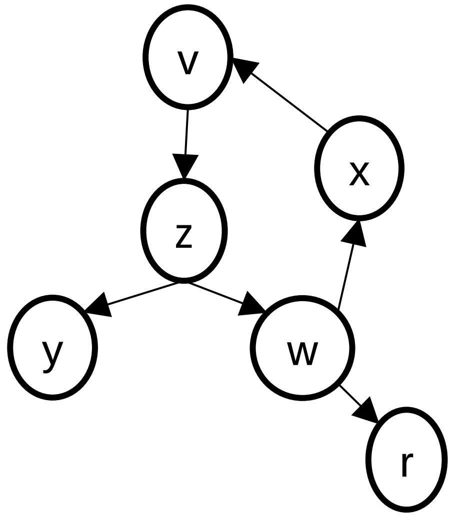
- $\Rightarrow$: Suppose v and w are in the same DFS-tree of G^T
  - Suppose r is the root of this tree
  - Since $r \rightarrow v$ in G^T, it must hold that $v \rightarrow r$ in G
  - Because of the order of the second DFS: post(r)>post(v) in G
  - Thus, there must be a path $r \rightarrow v$ in G: Otherwise, r had been visited last after v in G and thus 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)



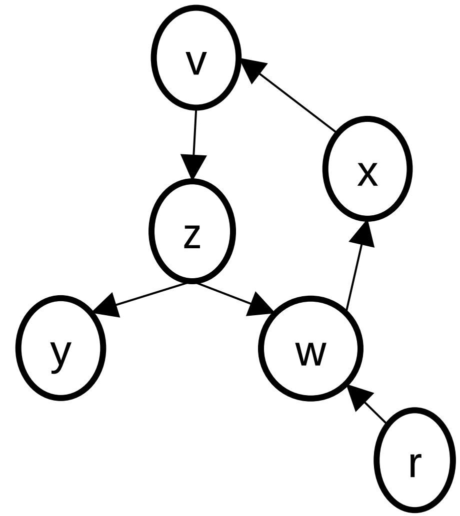
Examples (p() = post-order())



- $v \rightarrow w$
- 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: $p(w)>p(v)$
- w has higher p-value, 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 – needs to be done O(|V|) times
- Together: O(|V|+|E|)
- Since we need to look at each edge and node at least once to decide upon SCC, the problem is in Ω(|V|+|E|)
- There are faster algorithms that manage to compute SCCs in one traversal
  - Tarjan's algorithm, Gabow's algorithm