Algorithms and Data Structures

All Pairs Shortest Paths

Ulf Leser

Content of this Lecture

- All-Pairs Shortest Paths
  - Transitive closure: Warshall's algorithm
  - Shortest paths: Floyd's algorithm
- Reachability in Trees

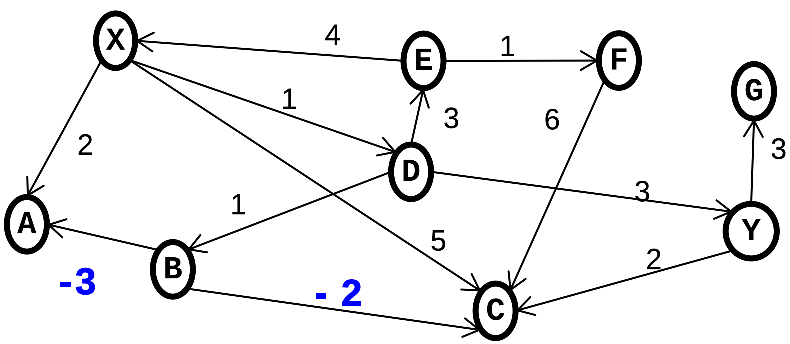
Shortest Path Problems

- Given a weighted digraph G

- Dijkstra finds the shortest path between a given start node and all other nodes for the case that all edge weights are positive
- All-pairs shortest paths: Given a digraph G with positive or negative edge weights, find the distance between all pairs of nodes

All-Pairs Shortest Paths: General Case

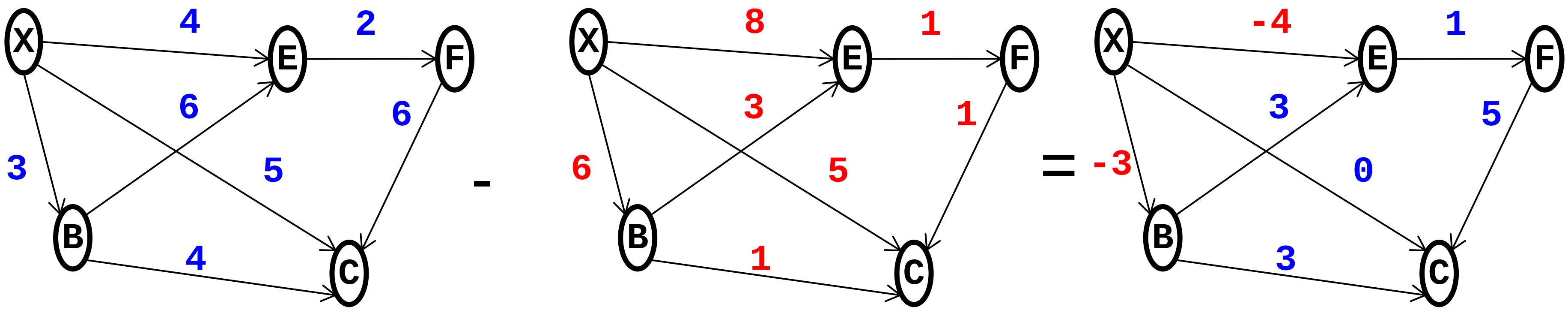
- Transitive closure with distances
- Result is $O(|V|^2)$ space, so don't try this for large graphs



→	A	B	C	D	E	F	G	X	Y
A	na	na	na	na	na	na	na	na	na
B	-3	na	-2	na	na	na	na	na	na
C	na	na	na	na	na	na	na	na	na
D	-2	1	...						
E									
F									
G									
X									
Y									

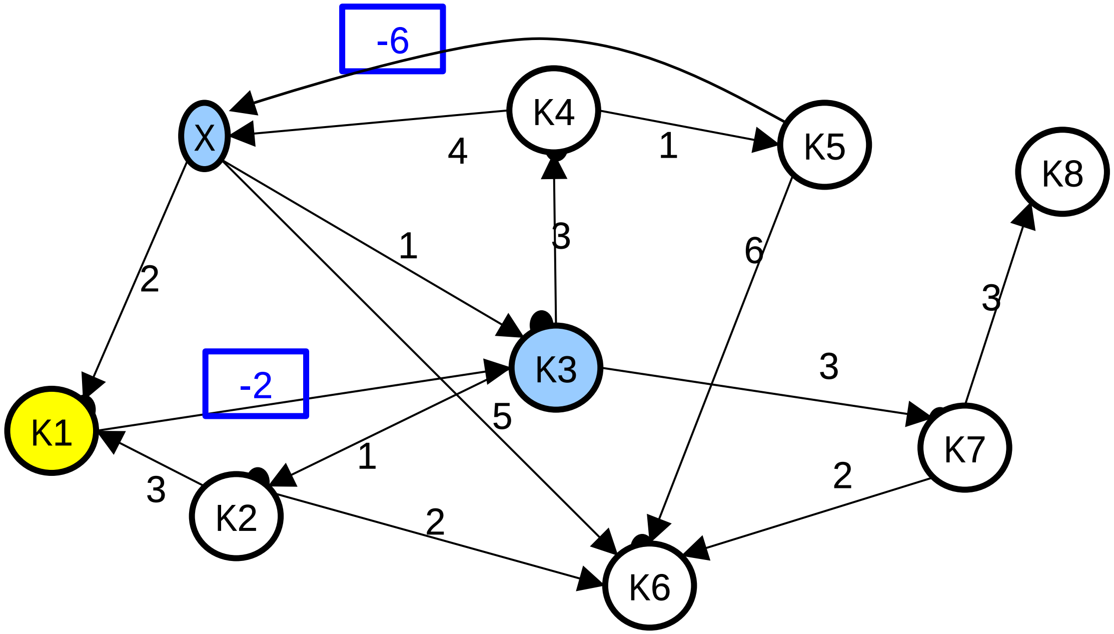
Why Negative Edge Weights?

- One application: 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

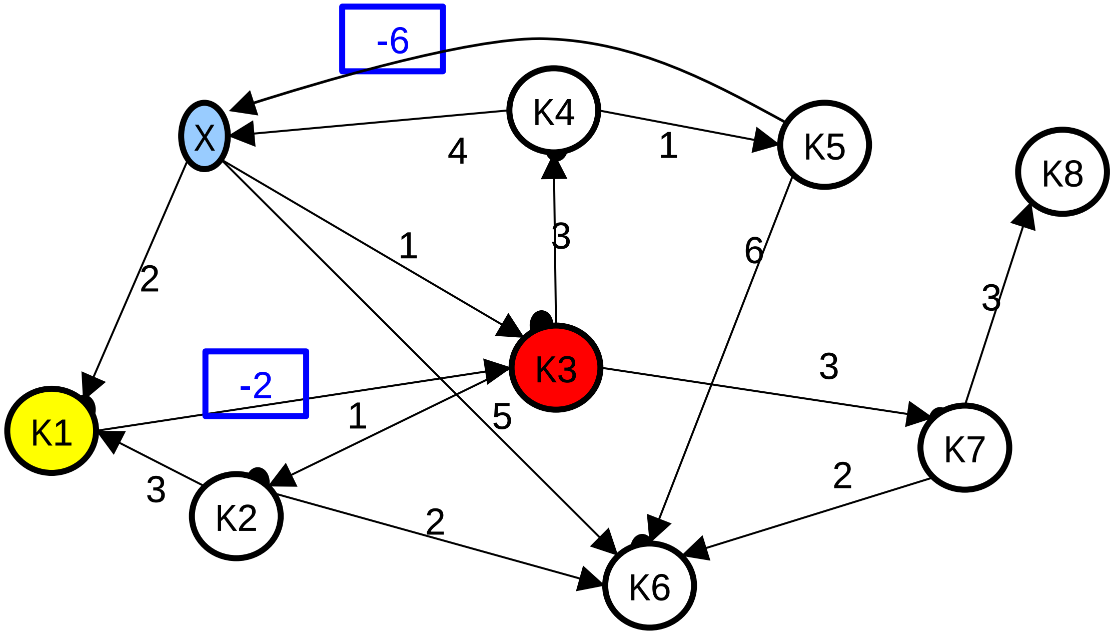
- Dijkstra's algorithm does not work
  - Recall that Dijkstra enumerates nodes by their shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it "shorter" (by negative edge weights)



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

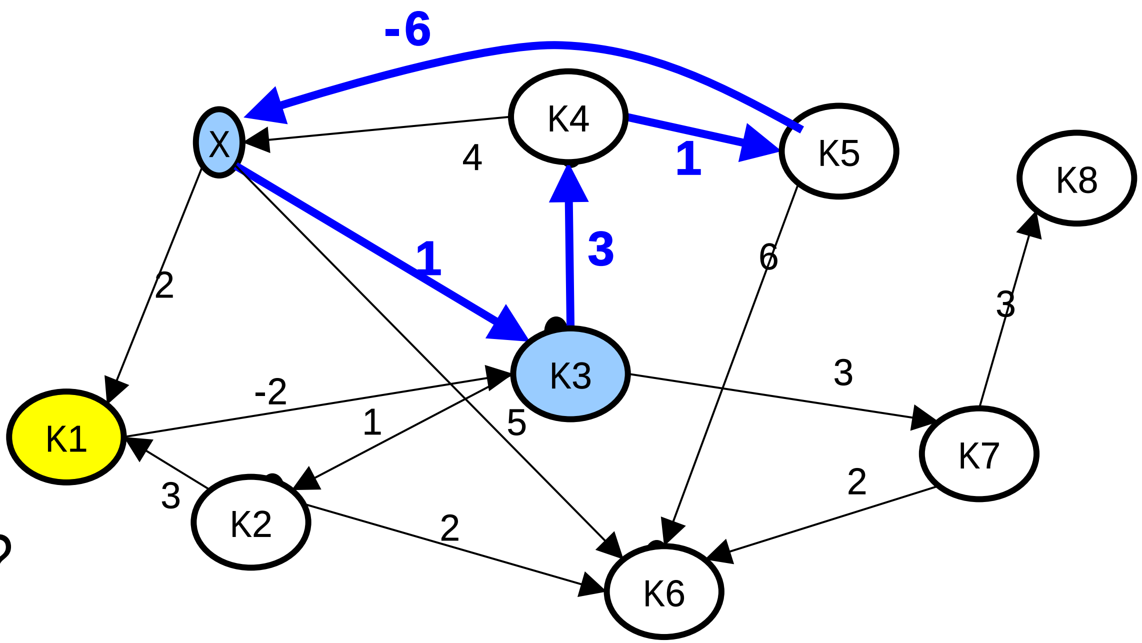
No Dijkstra

- Dijkstra's algorithm does not work
  - Recall that Dijkstra enumerates nodes by their 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

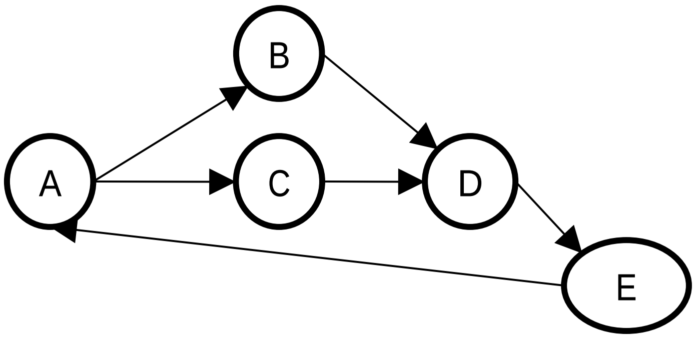


- Shortest path between X and K5?
  - X-K3-K4-K5: 5
  - X-K3-K4-K5-X-K3-K4-K5: 4
  - X-K3-K4-K5-X-K3-K4-K5-X-K3-K4-K5: 3
  - ...
- SP-Problem undefined if G contains a negative cycle

All-Pairs: First Approach

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

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

Naïve Algorithm

z appears nowhere; it is there to ensure that we stop when the longest possible shortest paths has been 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 $\leq z$
- Loops i and j look at all pairs reachable by a path of length $\leq z+1$
- Loop k extends path of length $\leq z$ by all outgoing edges
- Obviously $O(n^4)$

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		
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
- Computing TC 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
  - Computes $M*M$, M^2*M, M^3*M, ... $M^{n-1}*M$

Idea for Improvement

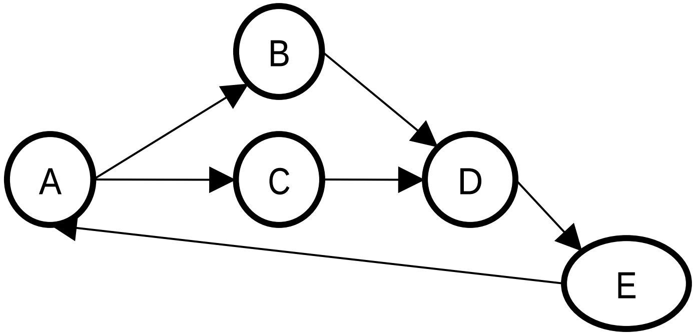
- Why not extend paths by all paths found so-far?
  - We compute
    $M^{2'}=M*M$: Path of length ≤ 2
    $M^{3'}=M^{2'}*M \cup M^{2'}*M^{2'}$: Path of length $\leq 2+1$ and $\leq 2+2$
    $M^{4'}=M^{3'}*M \cup M^{3'}*M^{2'} \cup M^{3'}*M^{3'}$, lengths $\leq 4+1$, $\leq 4+2$, $\leq 4+3/4$
    ...
    $M^{n'}= \ldots \cup M^{n-1'}*M^{n-1'}$
  - [We will implement it differently]
- Trick: We can stop much earlier
  - The longest shortest path can have length 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 M$^{2'}$, M$^{3'}$ ...
- 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 $\leq 2^z+2^z=2^{z+1}$
- Analysis: $O(n^3*\log(n))$
- But ... we can be even faster

Example – After z=1, 2, 3



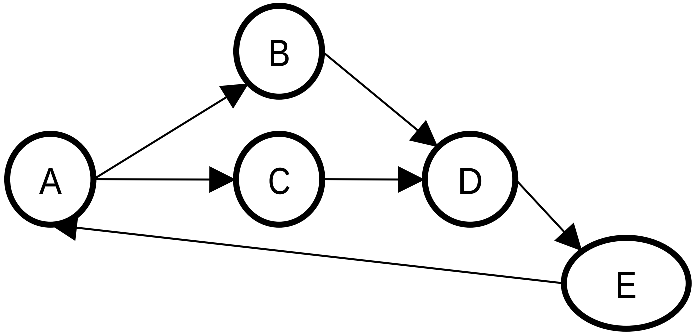
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	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

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 redundant 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.*

- Preparations
  - Fix an arbitrary order on nodes and assign each node its rank as ID
  - Let P_t be the set of all paths that contain only nodes with ID<t+1
- Idea: Compute P_t inductively
  - We start with P_1
  - We compute P_t, t>1, based on the assumption that P_{t-1} is known
  - We are done once t=n
- Induction
  - Suppose we know P_{t-1}
  - If we increase t by one, we admit one additional node, i.e., t
  - Now, every new path must have the form $x \rightarrow t \rightarrow y$
    - Paths with all IDs <t are already known
    - Node t is the only new player, must be in all new paths

Algorithm

- Enumerate paths by the IDs of the nodes they are allowed to contain
- t gives the highest allowed node ID inside 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;
```

Example – Warshall's Algorithm

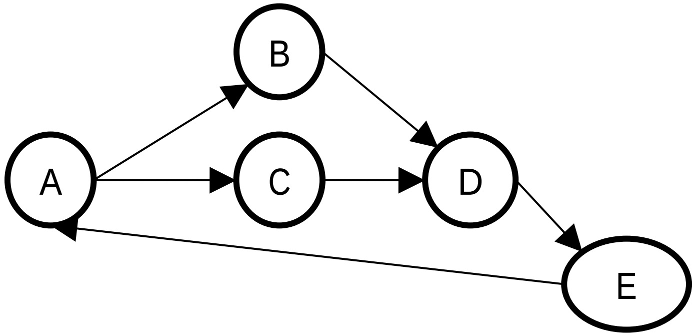
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

A allowed

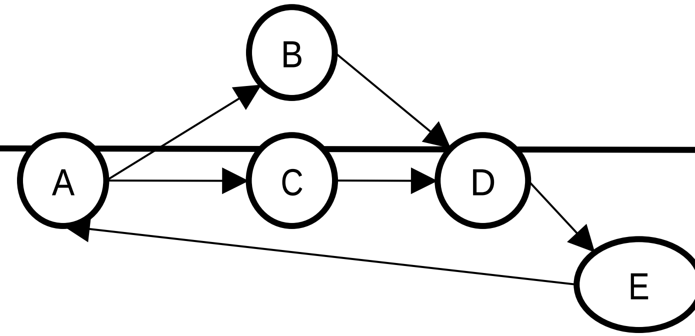
Connect
E-A with
A-B, A-C

maxID=A

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



Example – After t=A,B,C,D,E



t=„A“

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

B allowed

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

t=„B“

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

C allowed

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

No news

t=„C“

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

D allowed

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

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

E allowed

Connect
everything
with
everything

	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

Little change – Notable 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;
```

Drop z-Loop
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

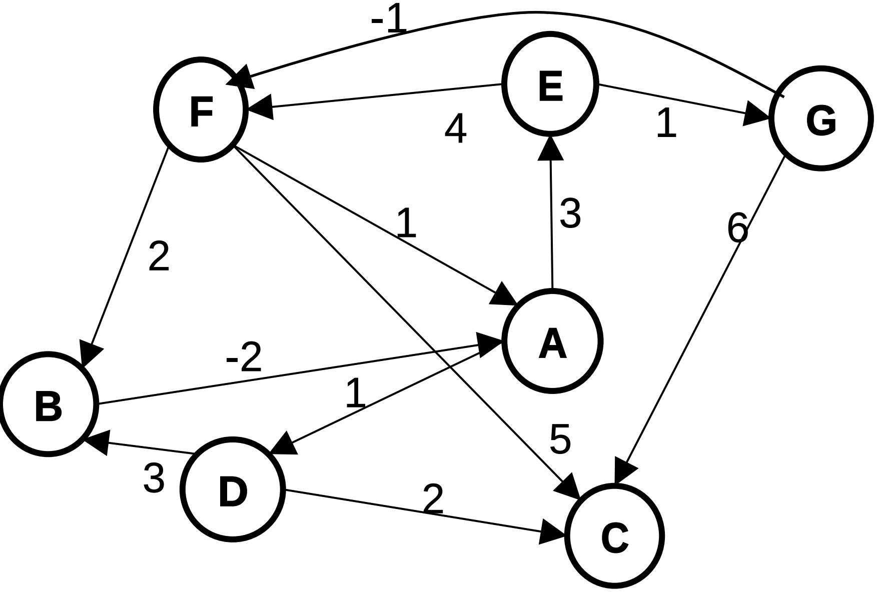
- All-Pairs Shortest Paths
  - Transitive closure: Warshall’s algorithm
  - Shortest paths: Floyd’s algorithm
- Reachability in Trees

Shortest Paths

Floyd, R. W. (1963). Algorithm 97: Shortest Path. *Communications of the ACM 5(6): 345.*

- Shortest paths: We need to compute the distance between all pairs of reachable nodes
- We use the same idea as Warshall: 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	

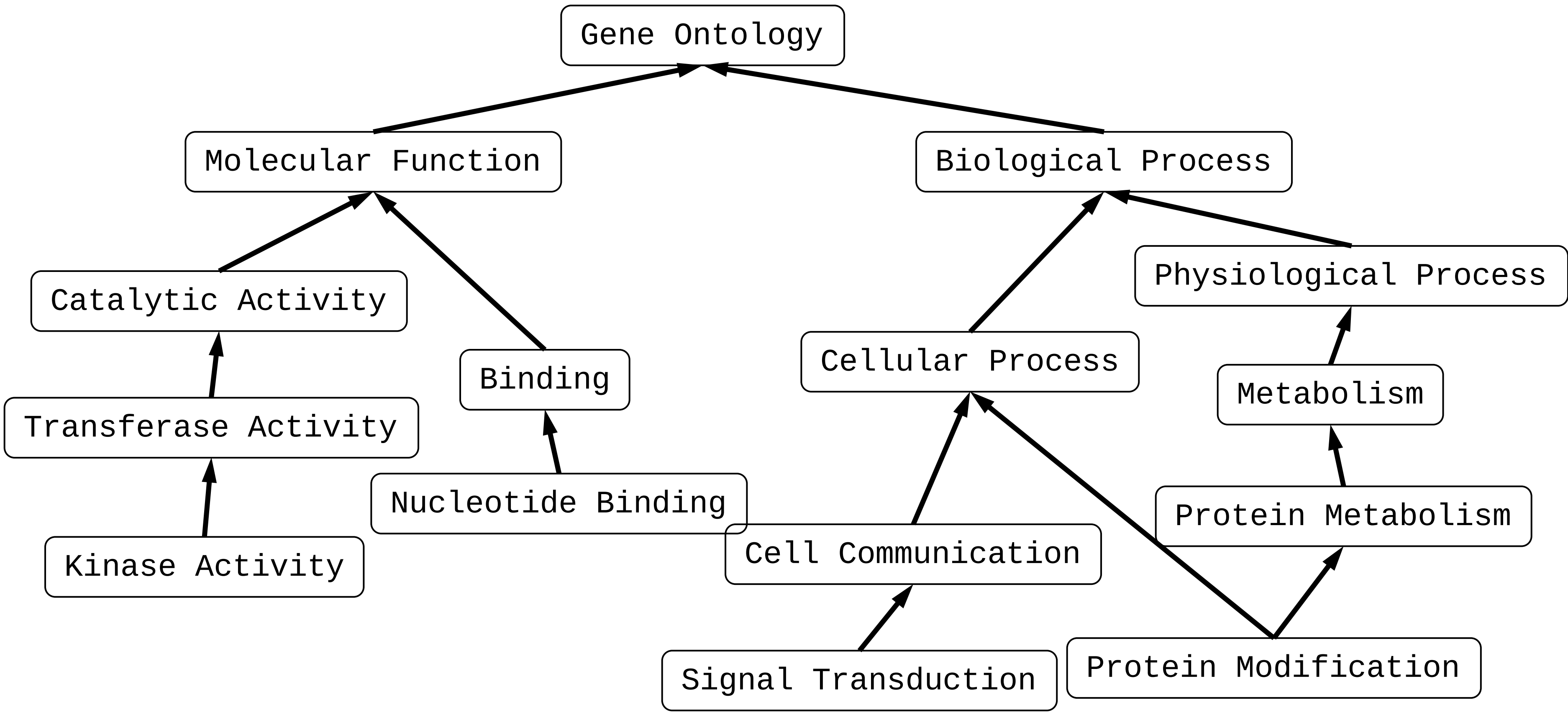
Summary (n=|V|, m=|E|)

- Warshall‘s algorithm computes the transitive closure of any unweighted digraph G in $O(n^3)$
- Floyd‘s algorithm computes the distances between any pair of nodes in a digraph without negative cycles in $O(n^3)$
- Johnson’s alg. solves the problem in $O(n^2*\log(n)+n*m)$
  - Which is faster for sparce graphs
- Storing both information requires $O(n^2)$
- Problem is easier for ...
  - Undirected graphs: Connected components
  - Graphs with only positive edge weights: All-pairs Dijkstra
  - Trees: Test for reachability in O(1) after O(n) preprocessing

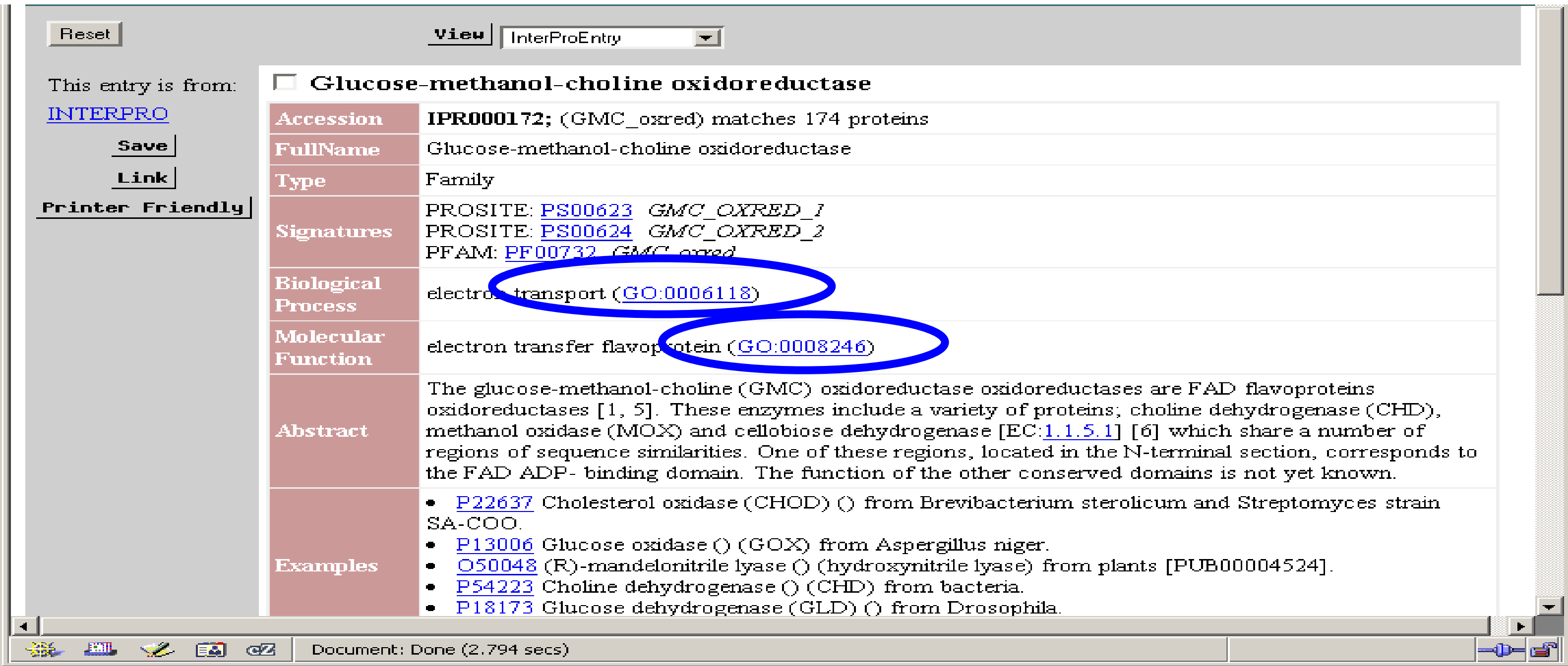
Content of this Lecture

- All-Pairs Shortest Paths
  - Transitive closure: Warshall's algorithm
  - Shortest paths: Floyd's algorithm
- Reachability in Trees

Gene Ontology – Describing Gene Function



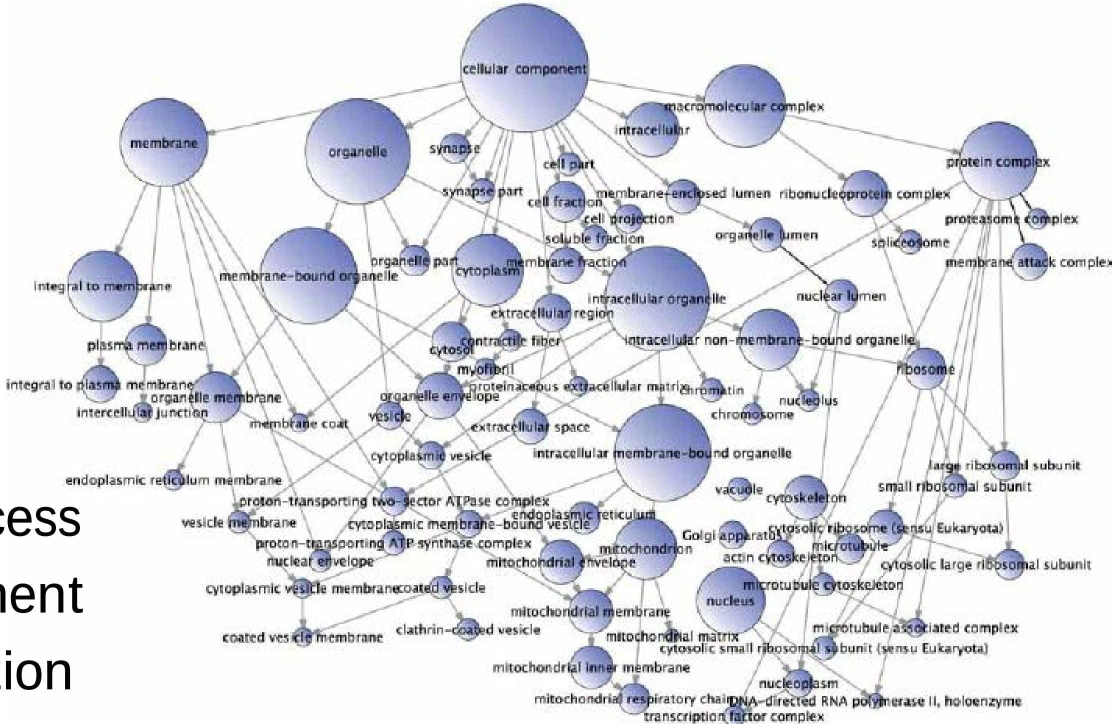
Database Annotation InterPro



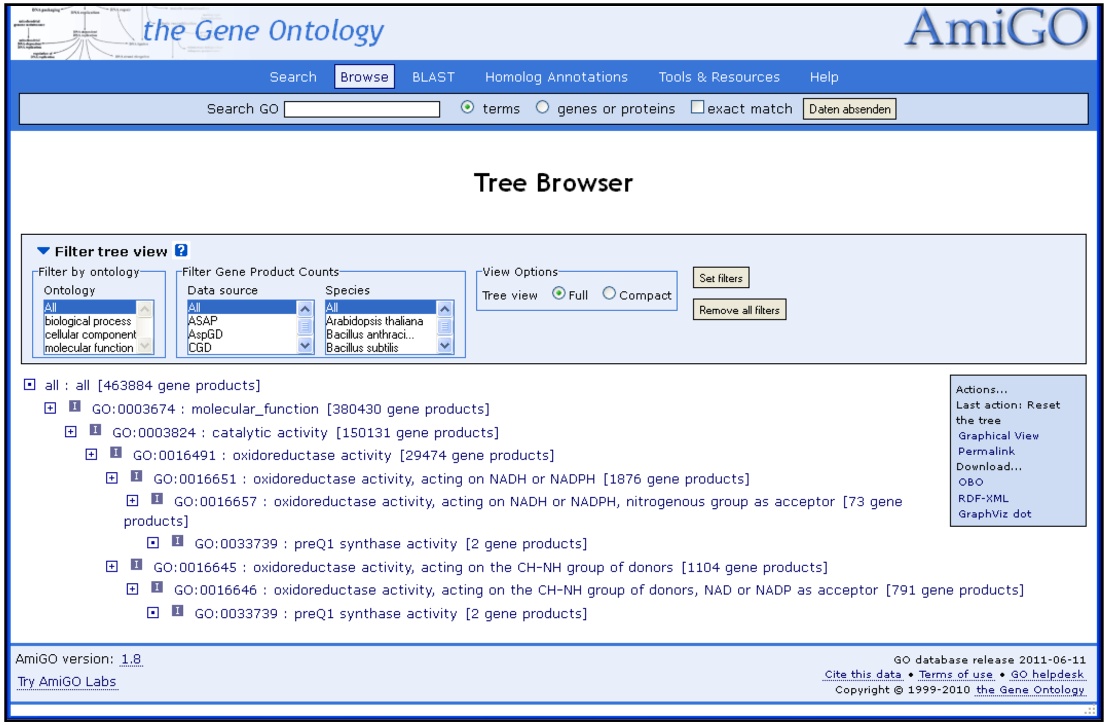
- Used by many databases
- Allows cross-database search
- Provides fixed meaning of terms
  - As informal textual description, not as formal definitions

A Large Ontology

- As of 10.6.2011
  - 34253 terms
  - 20831 biological process
  - 2844 cellular component
  - 9019 molecular function
  - 1559 obsolete terms
- Depth: >30
- Today: More than 40000 terms



Problem



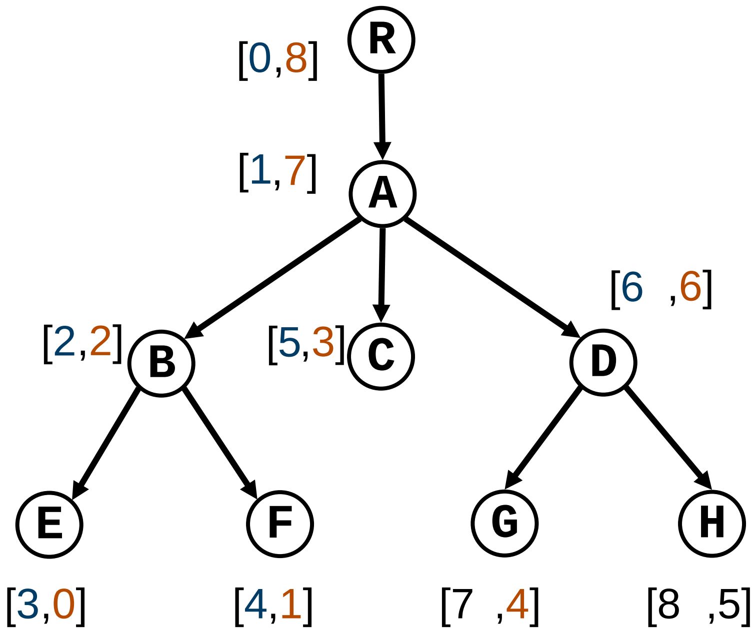
- To see whether a term X ISA term Y, we need to check whether Y lies on the path from root to X
- Reachability problem

Reachability in Trees

- Let T be a directed tree. A node v is reachable from a node w iff there is a path from w to v
- Testing reachability requires finding paths
  - Which is simple in trees
- Path length is bound by the length of the longest path, i.e., the depth of the tree
- This means O(n) in worst-case
- Let's see whether we can do this in constant time

Pre-/Postorder Numbers

- 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

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

- 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)