



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

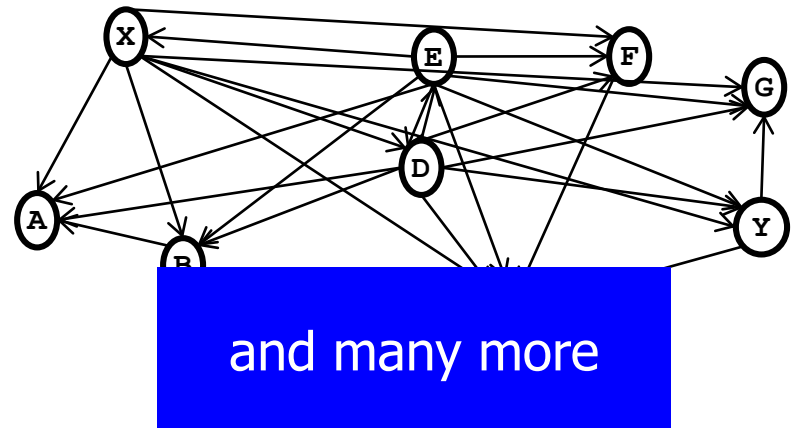
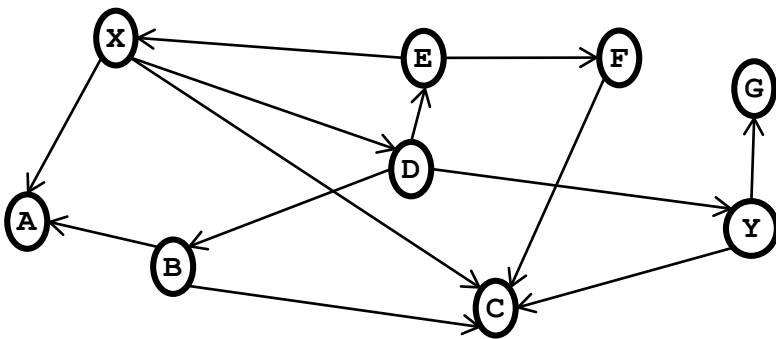
Recall: DFS

- We put **every node exactly once** on the stack
 - Once visited, never visited again
- We look at **every edge exactly once**
 - Outgoing edges of a visited node are never considered again
- U can be implemented as bit-array of size $|V|$, allowing $O(1)$ operations
 - Add, remove, getNextUnseen
- Altogether: **$O(n+m)$**

```
func void traverse (G, v node,
                  U set) {
    t := new Stack();
    t.put( v);
    U := U \ {v};
    while not t.isEmpty() do
        n := t.pop();
        print n;
        c := n.outgoingNodes();
        foreach x in c do
            if x∈U then
                U := U \ {x};
                t.push( x);
            end if;
        end for;
    end while;
}
```

Recall: Transitive Closure

- Definition
*Let $G=(V,E)$ be a digraph and $v_i, v_j \in V$. The **transitive closure** of G is a graph $G'=(V, E')$ where $(v_i, v_j) \in E'$ iff G contains a path from v_i to v_j .*
- TC usually is dense and represented as adjacency matrix
- Compact encoding of **reachability information**

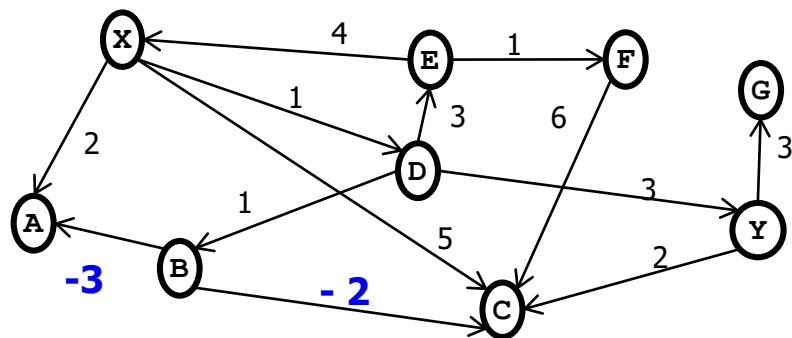


Shortest Path Problems

- 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

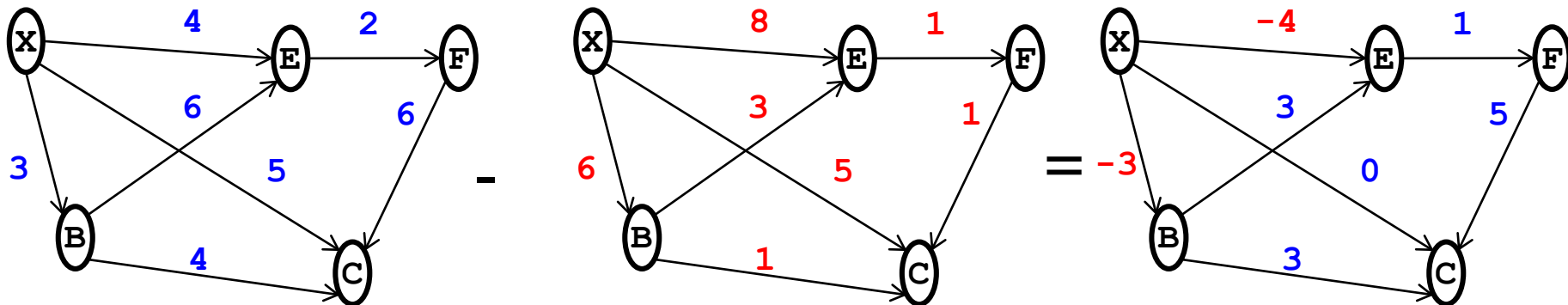
- **Transitive closure** with distances
- Result is $O(|V|^2)$ space, so don't try this for large graphs



→	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									

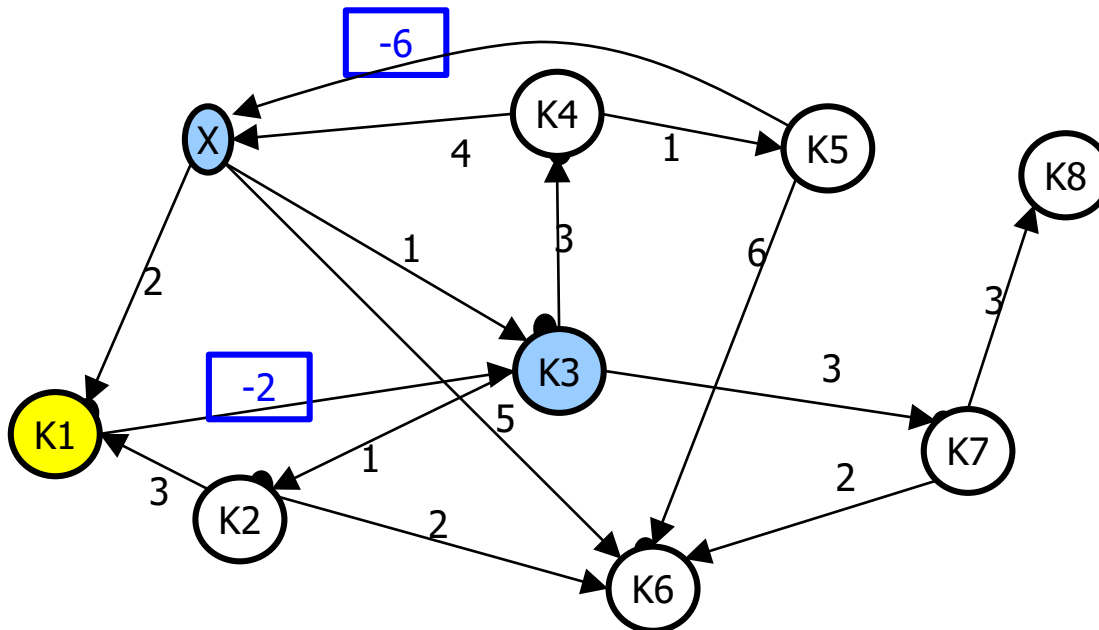
Why Negative Edge Weights?

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



No Dijkstra

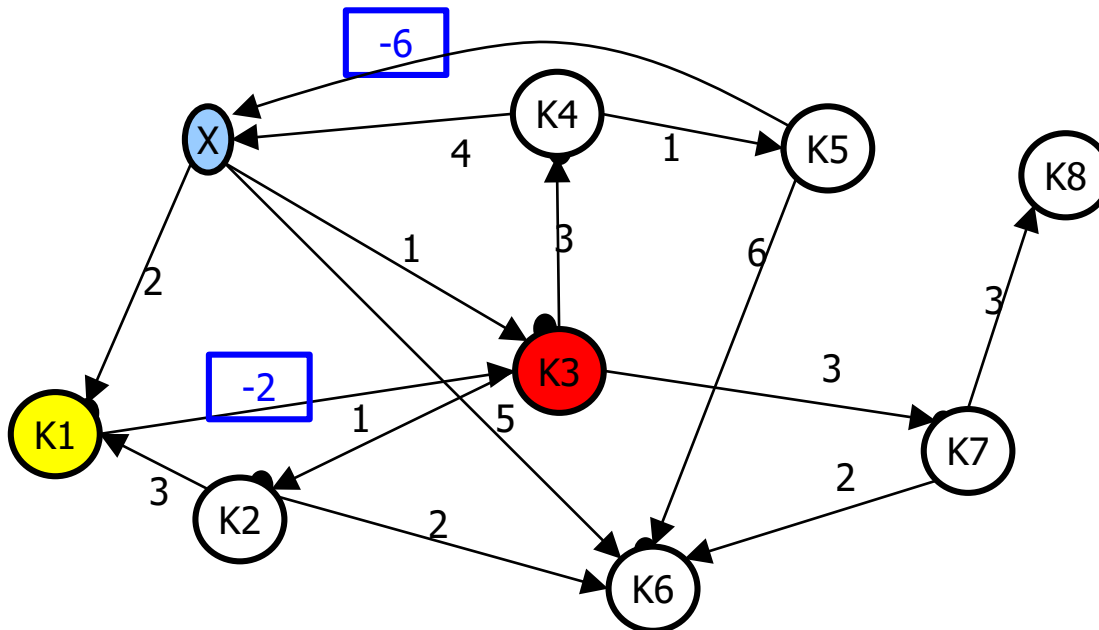
- Dijkstra's algorithm does not work
 - Recall that Dijkstra enumerates nodes by their shortest paths
 - Now: Adding a subpath to a so-far shortest path may make it "shorter" (by negative edge weights)



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

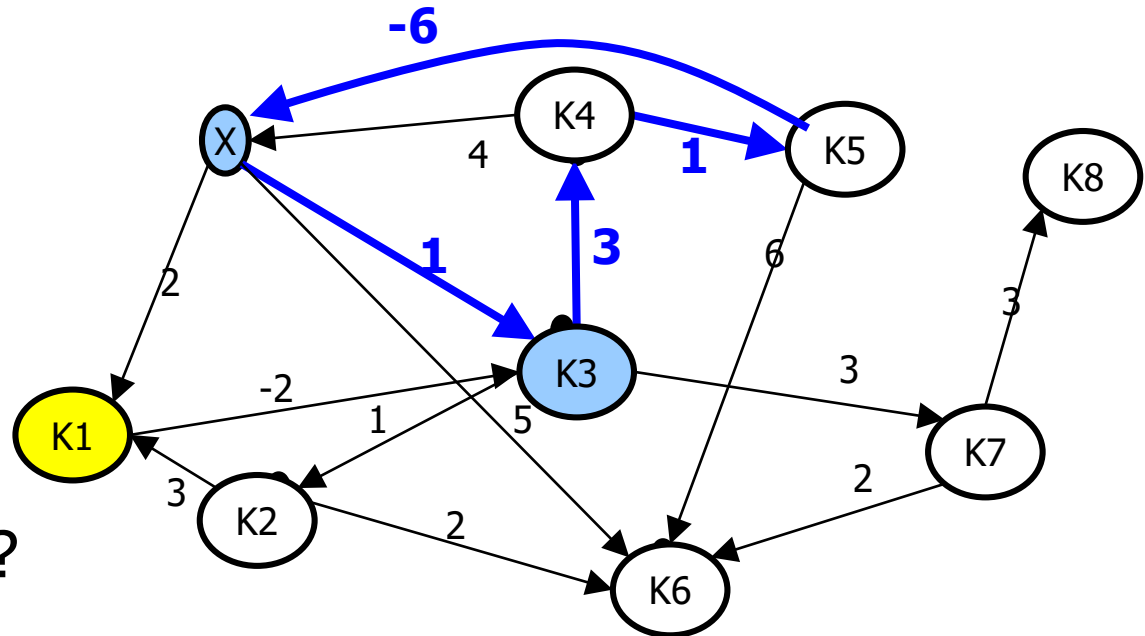
No Dijkstra

- 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	2
K2	2
K3	0
K4	4
K5	
K6	5
K7	4
K8	

Negative Cycles

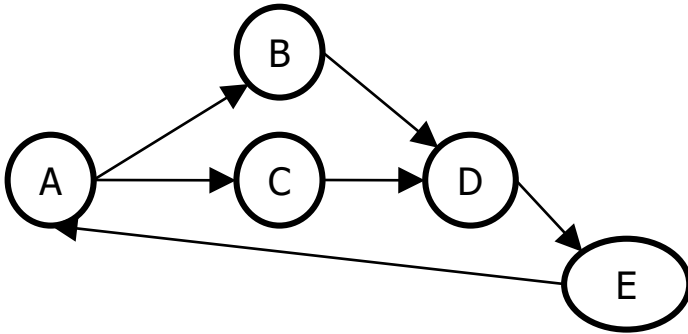


- Shortest path between X and K5?
 - X-K3-K4-K5: 5
 - X-K3-K4-K5-X-K3-K4-K5: 4
 - X-K3-K4-K5-X-K3-K4-K5-X-K3-K4-K5: 3
 - ...
- SP-Problem undefined if G contains a **negative cycle**

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: $x \xrightarrow{p_1} y \wedge y \xrightarrow{p_2} z \Rightarrow x \xrightarrow{p_1} y \xrightarrow{p_2} z = x \rightarrow z$
 - 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 than $|V|$
 - Or it would contain a cycle
- In each step, we store “reachable by a path of length $\leq k$ ” 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

```
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;
```

z appears nowhere; it is there to ensure that we stop when the **longest possible shortest paths** has been found

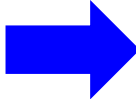
- M is the adjacency matrix of G, M'' eventually the TC of G
- M': Represents paths $\leq z$
- M'': Represents paths $\leq z+1$
- Reachability is transitive:
 $i \xrightarrow{p_1} j \wedge j \xrightarrow{p_2} k \Rightarrow i \xrightarrow{p_1} j \xrightarrow{p_2} k$
- Loops i and j look at all pairs reachable 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				

 \times

	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 $M \times M$** , 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 $M \times M$, $M^2 \times M$, $M^3 \times M$, ... $M^{n-1} \times M$

Idea for Improvement

- Why not extend paths **by all paths found so-far?**
 - We compute
$$M^{2'} = M \times M: \text{Path of length } \leq 2$$
$$M^{3'} = M^{2'} \times M \cup M^{2'} \times M^{2'}: \text{Path of length } \leq 2+1 \text{ and } \leq 2+2$$
$$M^{4'} = M^{3'} \times M \cup M^{3'} \times M^{2'} \cup M^{3'} \times M^{3'}, \text{ lengths } \leq 4+1, \leq 4+2, \leq 4+3/4$$

...

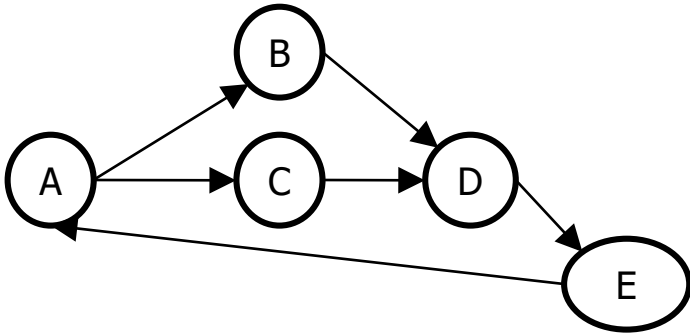
$$M^{n'} = \dots \cup M^{n-1'} \times M^{n-1'}$$
 - [We will implement it differently]
- Trick: We can **stop much earlier**
 - The longest shortest path can have length at most n
 - Thus, it suffices to compute $M^{\log(n)'} = \dots \cup M^{\log(n)'} \times M^{\log(n)'}$

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^z} , $M^{2^{z+1}}$...
- 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: $O(n^3 \cdot \log(n))$
- 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

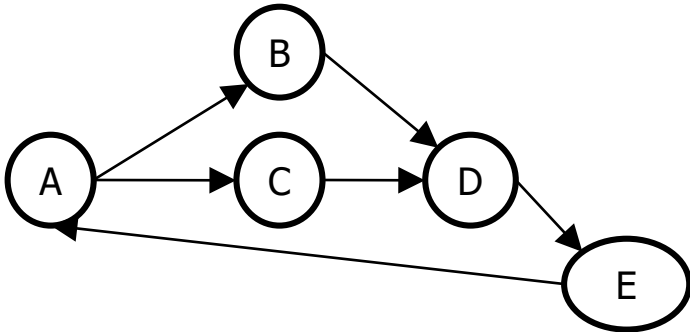
Path length:

≤ 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		

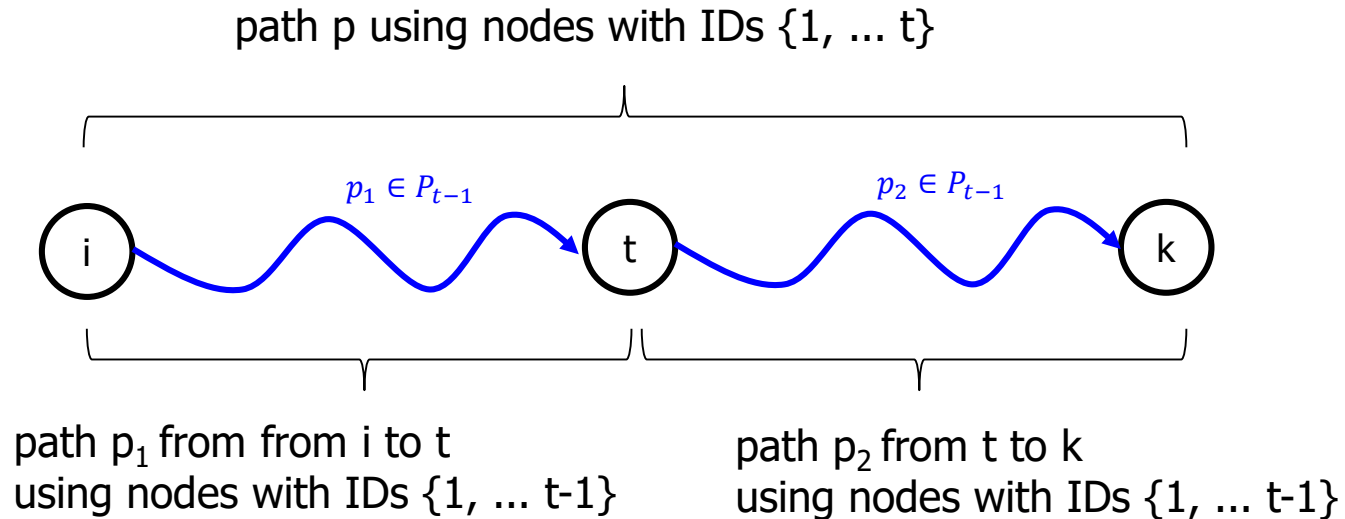
- Note: The path $A \rightarrow D$ is found twice: $A \rightarrow B \rightarrow D$ / $A \rightarrow C \rightarrow D$
- Can we stop “searching” $A \rightarrow D$ once we found $A \rightarrow B \rightarrow D$?
- Can we enumerate paths such that redundant paths are discovered less often (i.e., less paths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

Warshall's Algorithm

- 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
 - t gives the highest allowed **node ID inside a path**
- 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., ID t
 - Now, every **additional path** must have the form $i \xrightarrow{p_1 \in P_{t-1}} t \xrightarrow{p_2 \in P_{t-1}} k$
 - Paths with all IDs < t are already known
 - Node t is the only new player, must be in all new paths

Warshall's Algorithm

- Enumerate paths by the **IDs of the nodes they are allowed to contain**
- t gives the highest allowed node ID inside a path



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 $i \rightarrow t$ and $t \rightarrow k$
- For every such pair, we set the path $i \rightarrow k$ to 1

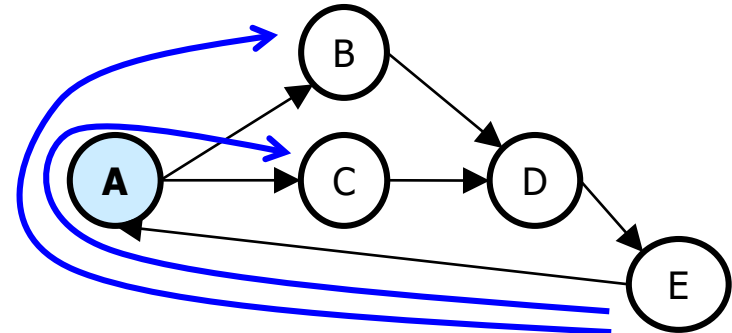
```
1. G = (V, E);
2. M := adjacency_matrix( G );
3. n := |V|;
4. for t := 1..n do
5.   for i = 1..n do
6.     if M[i,t]=1 then
7.       for k=1 to n do
8.         if M[t,k]=1 then
9.           M[i,k] := 1;
10.        end if;
11.       end for;
12.     end if;
13.   end for;
14. end for;
```

Example – Warshall's Algorithm

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

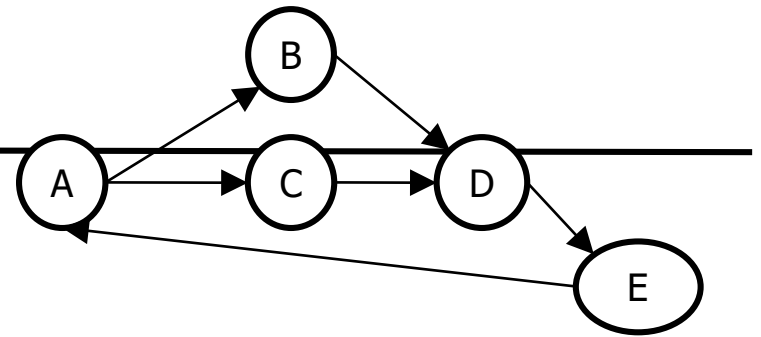
maxID=A

	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 $t=A,B,C,D,E$



$t=„A“$

	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1	<u>1</u>	<u>1</u>		

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

$t=„B“$

	A	B	C	D	E
A		1	1	<u>1</u>	
B				1	
C				1	
D					1
E	1	1	1	<u>1</u>	

C allowed
Connect
A-C/E-C
with C-D
No news

$t=„C“$

	A	B	C	D	E
A		1	1	<u>1</u>	
B				1	
C				1	
D					1
E	1	1	1	<u>1</u>	

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

	A	B	C	D	E
A		1	1	1	<u>1</u>
B				1	<u>1</u>
C				1	<u>1</u>
D					1
E	1	1	1	1	<u>1</u>

E allowed
Connect
**everything
with
everything**

	A	B	C	D	E
A	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	1
B	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	1
C	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	1
D	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	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;
```

$O(n^4)$



Drop z-
Loop
Swap i and
j loop
Rephrase j
into t

```
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;
```

$O(n^3)$

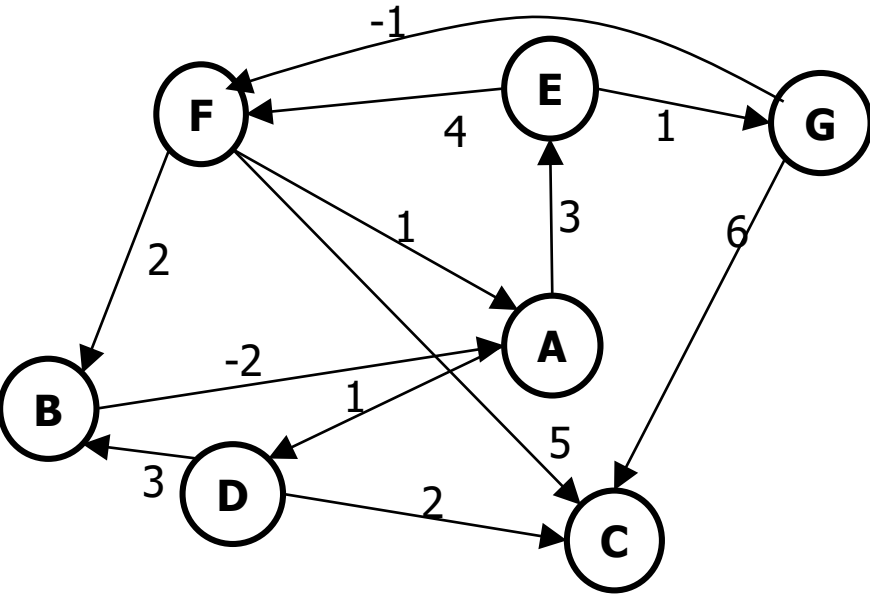
Content of this Lecture

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

Shortest Paths

- 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 with IDs smaller than t inside a path
 - 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 1/3



	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	

Example 2/3

	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	

	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	

	A	B	C	D	E	F	G
A	2	4	3	1	3	7	4
B	-2	2	1	-1	1	5	2
C							
D	1	3	2	2	4	8	5
E						4	1
F	0	2	3	1	3	7	4
G			6			-1	

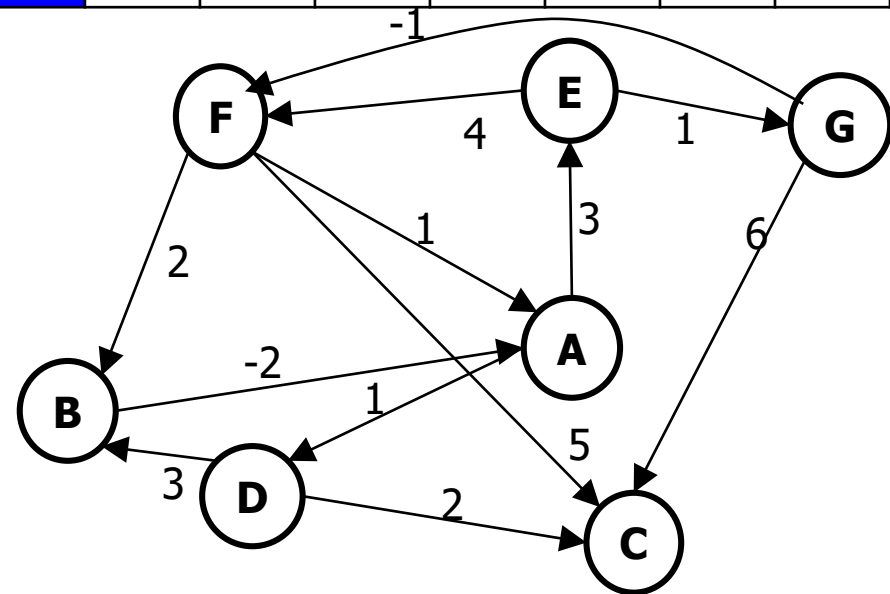
	A	B	C	D	E	F	G
A	2	4	3	1	<u>3</u>		
B	<u>-2</u>	2	1	-1	<u>1</u>		
C							
D	1	3	2	2	4		
E						4	1
F	<u>0</u>	<u>2</u>	3	1	<u>3</u>		
G			6			-1	

Example 3/3

	A	B	C	D	E	F	G
A	<u>2</u>	<u>4</u>	<u>3</u>	<u>1</u>	<u>3</u>	7	<u>4</u>
B	<u>-2</u>	<u>2</u>	<u>1</u>	<u>-1</u>	<u>1</u>	5	<u>2</u>
C							
D	<u>1</u>	<u>3</u>	<u>2</u>	<u>2</u>	<u>4</u>	8	<u>5</u>
E	4	6	7	5	7	4	1
F	0	2	3	1	3	7	4
G	-1	1	2	0	2	-1	3

	A	B	C	D	E	F	G
A	<u>2</u>	<u>4</u>	<u>3</u>	<u>1</u>	<u>3</u>	3	<u>4</u>
B	<u>-2</u>	<u>2</u>	<u>1</u>	<u>-1</u>	<u>1</u>	1	<u>2</u>
C							
D	<u>1</u>	<u>3</u>	<u>2</u>	<u>2</u>	<u>4</u>	4	<u>5</u>
E	0	2	3	1	3	0	1
F	<u>0</u>	<u>2</u>	<u>3</u>	<u>1</u>	<u>3</u>	3	<u>4</u>
G	-1	1	2	0	2	-1	3

	A	B	C	D	E	F	G
A	2	4	3	1	3	7	4
B	-2	2	1	-1	1	5	2
C							
D	1	3	2	2	4	8	5
E						4	1
F	0	2	3	1	3	7	4
G			6			-1	



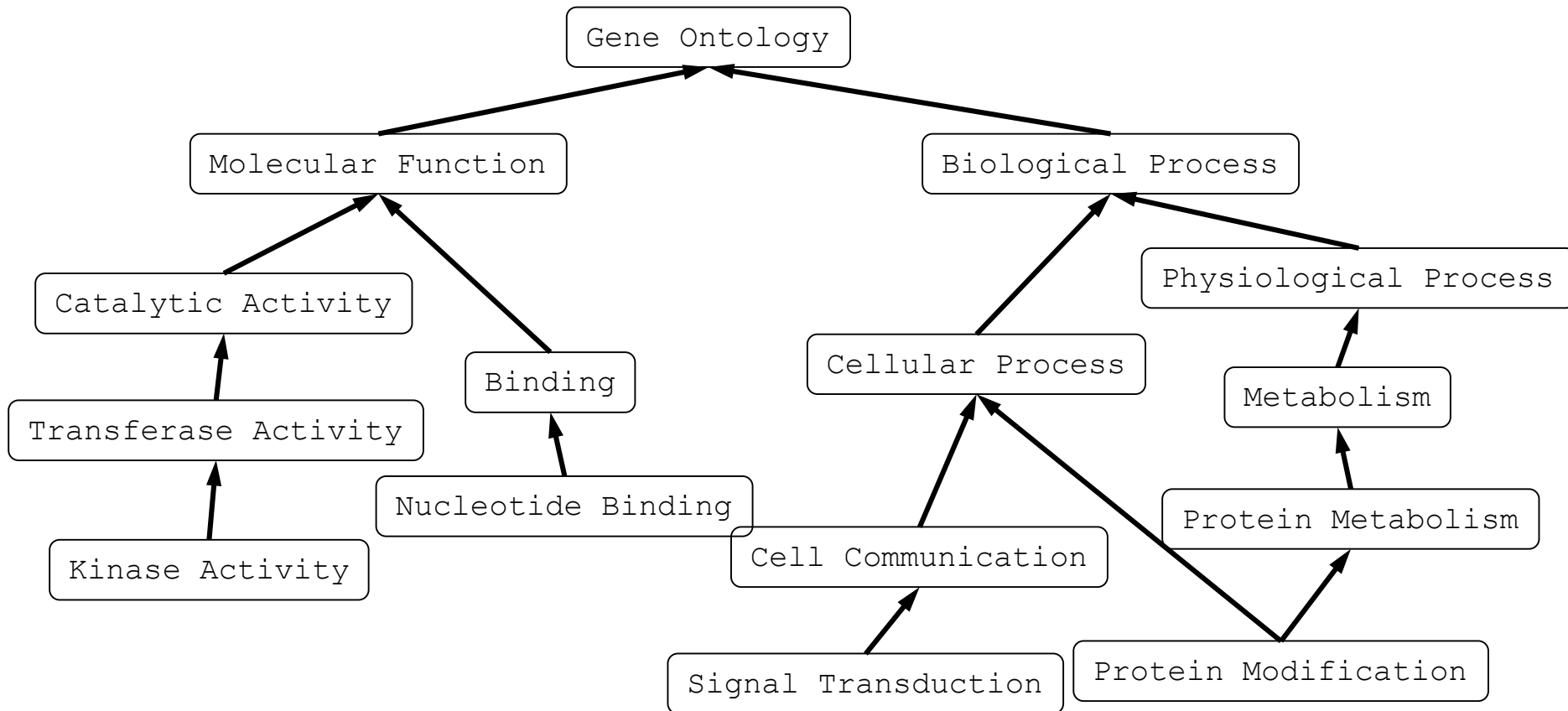
Summary ($n=|V|$, $m=|E|$)

- 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 $O(n^2 \cdot \log(n) + n \cdot m)$
 - Which is faster for sparse 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

Reset View InterProEntry

This entry is from: [INTERPRO](#)

Save Link Printer Friendly

Glucose-methanol-choline oxidoreductase

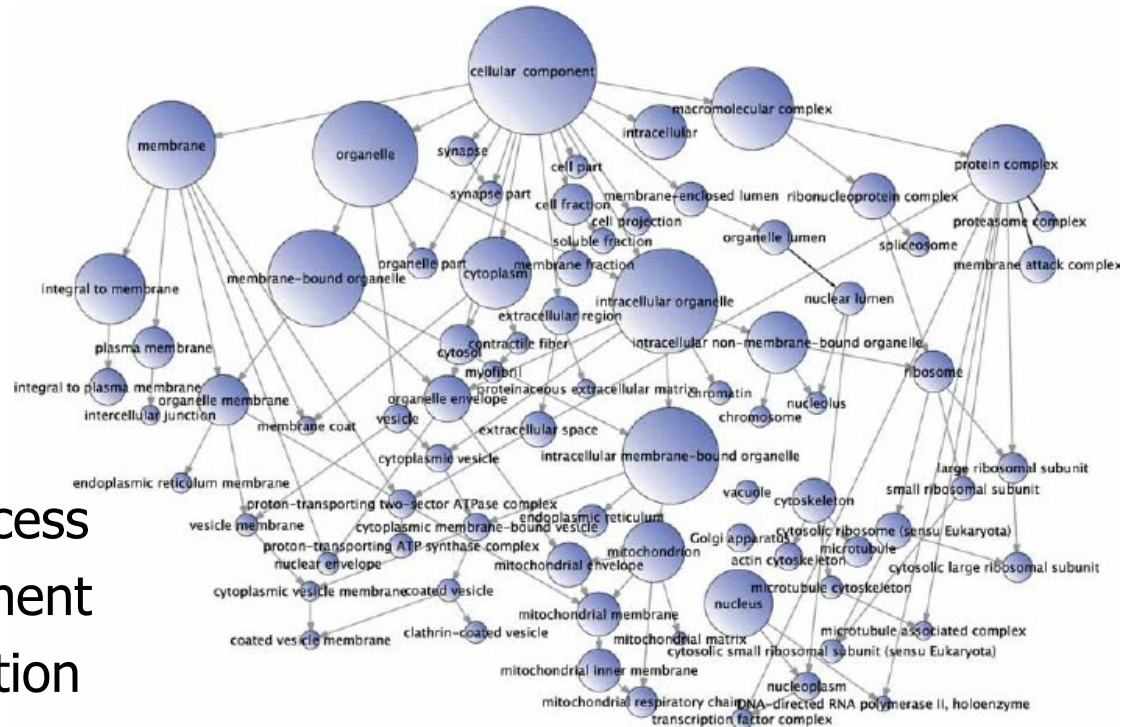
Accession	IPR000172; (GMC_oxred) matches 174 proteins
FullName	Glucose-methanol-choline oxidoreductase
Type	Family
Signatures	PROSITE: PS00623 GMC_OXRED_1 PROSITE: PS00624 GMC_OXRED_2 PFAM: PF00732 GMC_oxred
Biological Process	electron transport (GO:0006118)
Molecular Function	electron transfer flavoprotein (GO:0008246)
Abstract	The glucose-methanol-choline (GMC) oxidoreductase oxidoreductases are FAD flavoproteins oxidoreductases [1, 5]. These enzymes include a variety of proteins; choline dehydrogenase (CHD), methanol oxidase (MOX) and cellobiose dehydrogenase [EC:1.1.5.1] [6] which share a number of regions of sequence similarities. One of these regions, located in the N-terminal section, corresponds to the FAD ADP- binding domain. The function of the other conserved domains is not yet known.
Examples	<ul style="list-style-type: none">• P22637 Cholesterol oxidase (CHOD) () from Brevibacterium sterolicum and Streptomyces strain SA-COO.• P13006 Glucose oxidase () (GOX) from Aspergillus niger.• O50048 (R)-mandelonitrile lyase () (hydroxynitrile lyase) from plants [PUB00004524].• P54223 Choline dehydrogenase () (CHD) from bacteria.• P18173 Glucose dehydrogenase (GLD) () from Drosophila.

Document: Done (2.794 secs)

- 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

The screenshot displays the AmiGO Tree Browser interface. At the top, there is a search bar with the text "Search GO" and a "Daten absenden" button. Below the search bar, the "Tree Browser" title is centered. The main content area is divided into several sections:

- Filter tree view:** Includes a "Filter by ontology" dropdown menu with options like "biological process", "cellular component", and "molecular function".
- Filter Gene Product Counts:** Includes a "Data source" dropdown menu with options like "ASAP", "AspGD", and "CGD", and a "Species" dropdown menu with options like "Arabidopsis thaliana", "Bacillus anthraci...", and "Bacillus subtilis".
- View Options:** Includes a "Tree view" section with radio buttons for "Full" (selected) and "Compact", and buttons for "Set filters" and "Remove all filters".

The main tree view shows a hierarchical structure of Gene Ontology terms. The root node is "all : all [463884 gene products]". The tree is expanded to show the following structure:

- GO:0003674 : molecular_function [380430 gene products]
 - GO:0003824 : catalytic activity [150131 gene products]
 - GO:0016491 : oxidoreductase activity [29474 gene products]
 - GO:0016651 : oxidoreductase activity, acting on NADH or NADPH [1876 gene products]
 - GO:0016657 : oxidoreductase activity, acting on NADH or NADPH, nitrogenous group as acceptor [73 gene products]
 - GO:0033739 : preQ1 synthase activity [2 gene products]
 - GO:0016645 : oxidoreductase activity, acting on the CH-NH group of donors [1104 gene products]
 - GO:0016646 : oxidoreductase activity, acting on the CH-NH group of donors, NAD or NADP as acceptor [791 gene products]
 - GO:0033739 : preQ1 synthase activity [2 gene products]

At the bottom of the interface, there is a footer with the text "AmiGO version: 1.8" and "GO database release 2011-06-11".

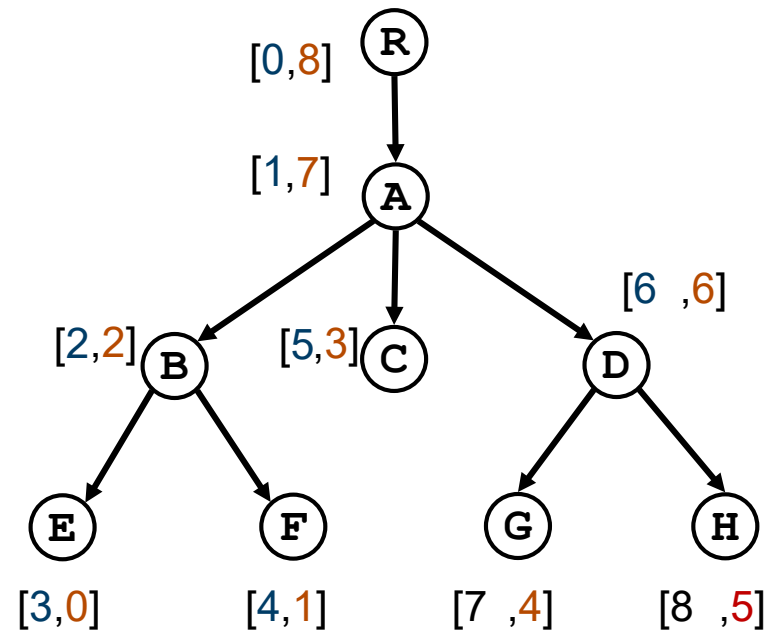
- To see whether a term X IS_A term Y, we need to check whether Y lies on the path from root to X
- Reachability problem

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 preprocess the data to 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 $O(1)$**

