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

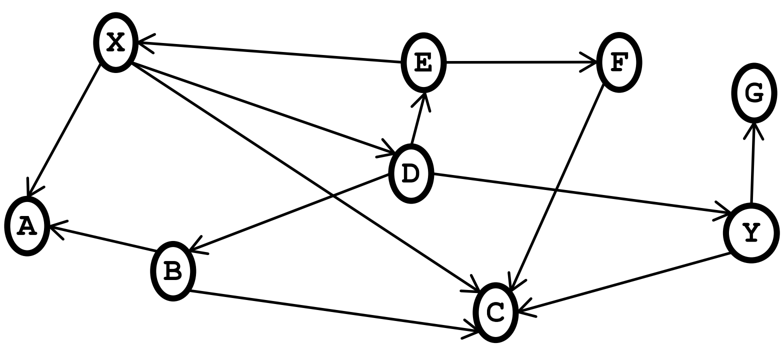
Recall: DFS

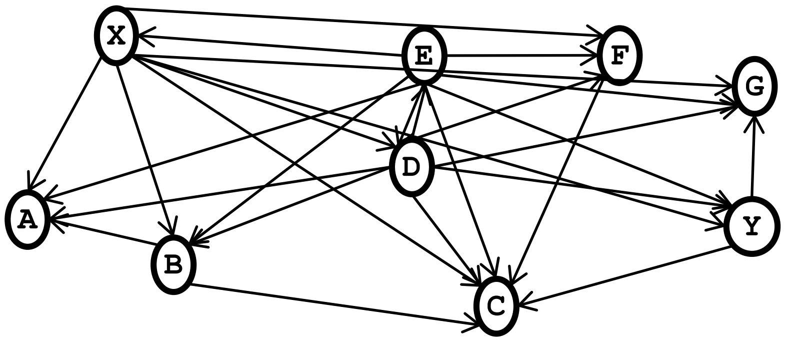
- We put every node exactly once on the stack
  - Once visited, never visited again
- We look at every edge exactly once
  - Outgoing edges of a visited node are never considered again
- U can be implemented as bit-array of size |V|, allowing O(1) operations
  - Add, remove, getNextUnseen
- Altogether: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

```
func void traverse (G graph,
                    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

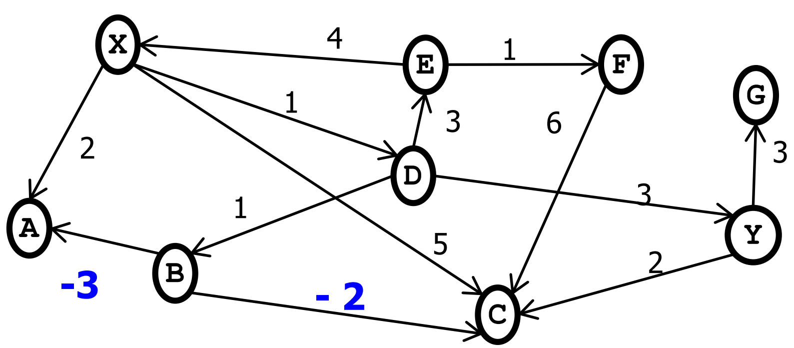




and many more

Shortest Path Problems

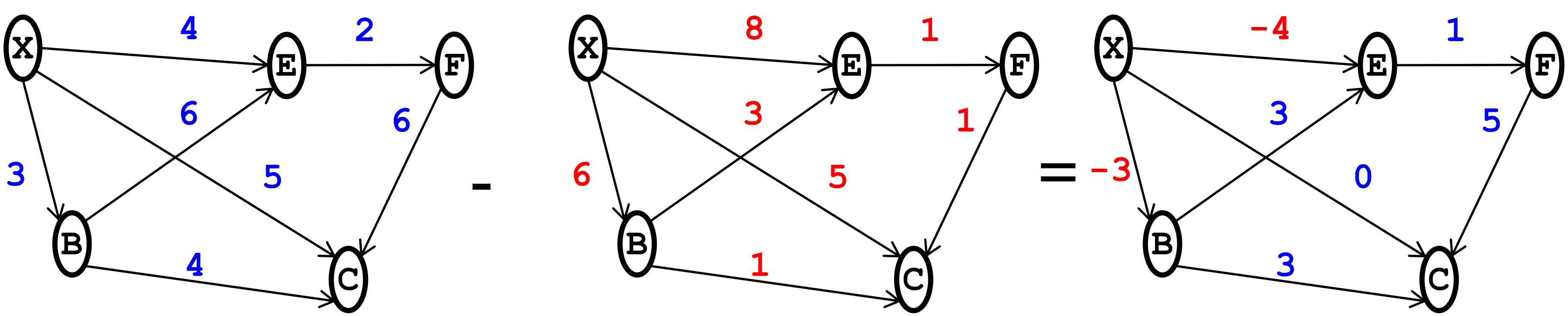
- Dijkstra finds shortest path between a given start node and all other nodes assuming that all edge weights are positive
- All-pairs shortest paths: Given a digraph G with positive or negative edge weights, find the (cycle-free) distance between all pairs of nodes
  - We will interpret "find" as "compute the distance matrix"



→	A	B	C	D	E	F	G	X	Y
A	-	-	-	-	-	-	-	-	-
B	-3	-	-2	-	-	-	-	-	-
C	-	-	-	-	-	-	-	-	-
D	-2	1	-1	-	3	4	6	7	3
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



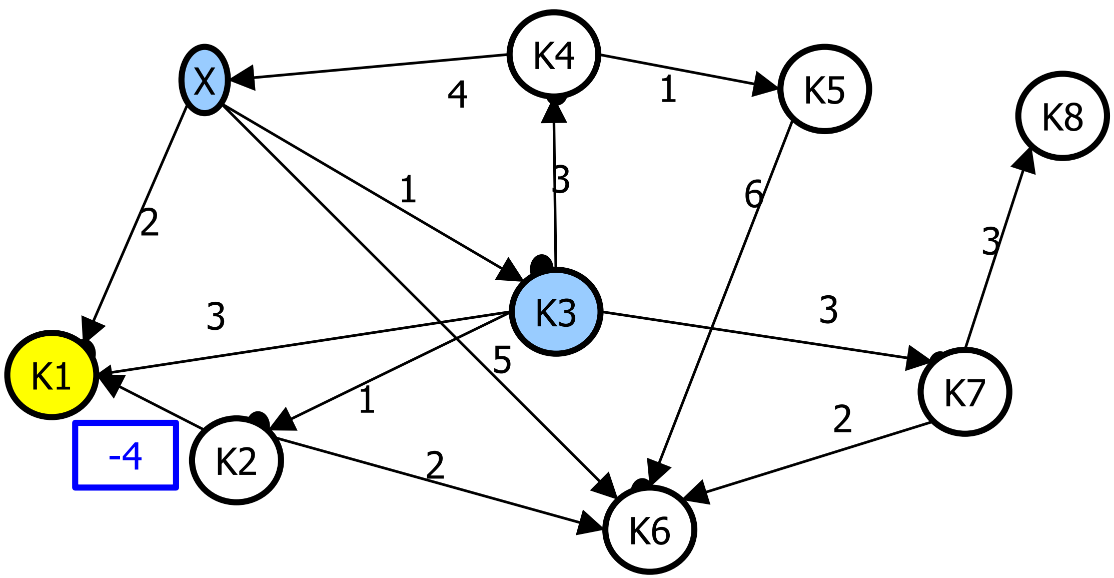
Cost

Incoming

Shortest path = max revenue

No Dijkstra

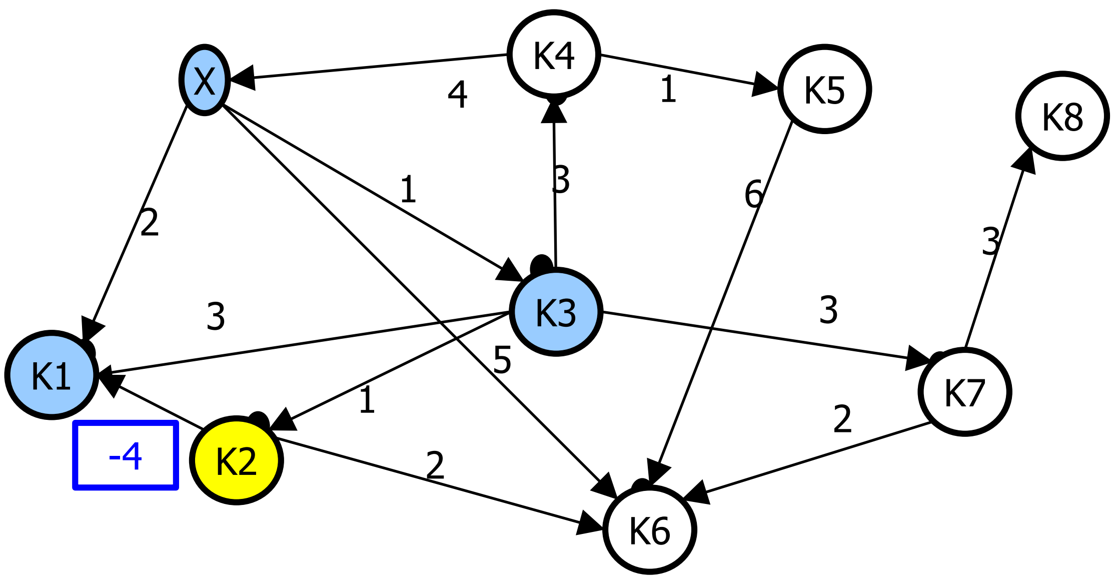
- Dijkstra’s algorithm does not work
  - Recall that Dijkstra enumerates nodes by their shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it “shorter” (by negative edge weights)



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

No Dijkstra

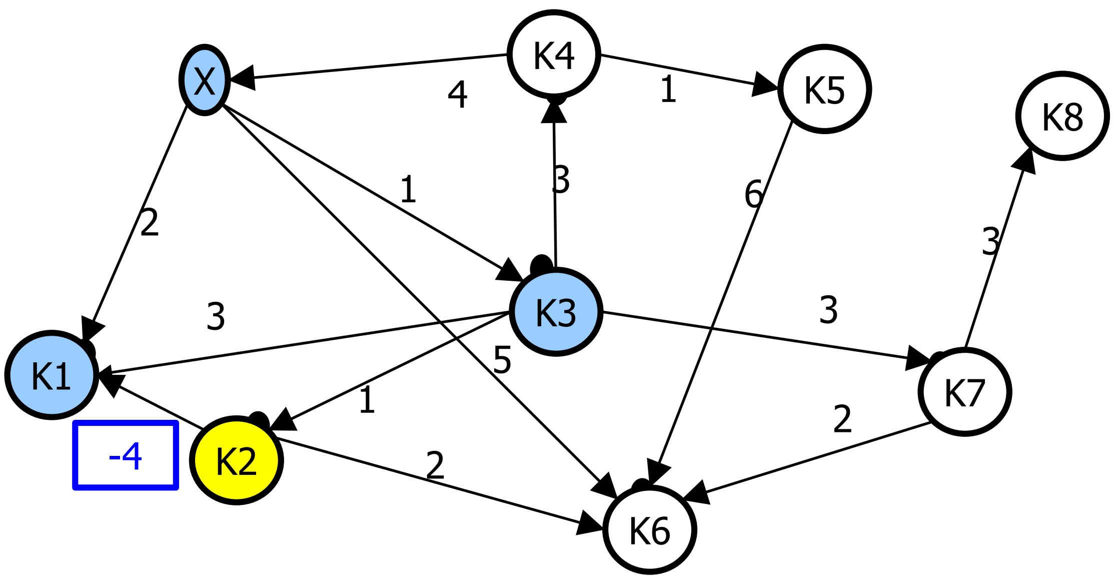
- Dijkstra’s algorithm does not work
  - Recall that Dijkstra enumerates nodes by their shortest paths
  - Now: Adding a subpath to a so-far shortest path may make it “shorter” (by negative edge weights)



X	0
K1	2
K2	2
K3	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)

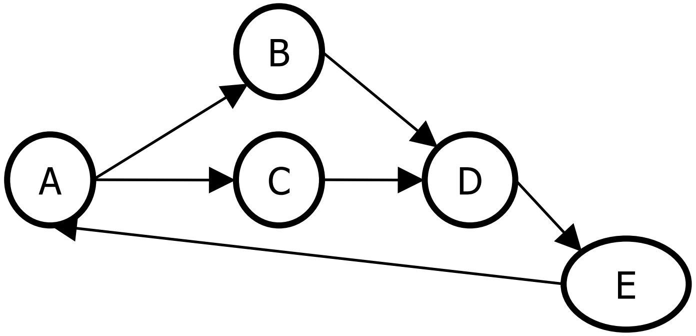


X	0
K1	2
K2	2
K3	1
K4	4
K5	
K6	5
K7	4
K8	

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: $x \overset{p_1}{\rightarrow} y \wedge y \overset{p_2}{\rightarrow} z \;\Rightarrow\; x \overset{p_1}{\rightarrow} y \overset{p_2}{\rightarrow} z = x \rightarrow z$
  - We may use this idea to iteratively build 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|-1
    - Or it would contain a cycle
- In each step, we store "reachable by a path of length ≤k" in a matrix

Example



	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

Path length: ≤2 ≤3 ≤4

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 ≤z
- M": Represents paths ≤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 ≤z+1
  - Loop k extends path of length ≤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				

x

	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 MxM, 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 MxM, M^2xM, M^3xM, … M^{n-2}xM

Idea for Improvement

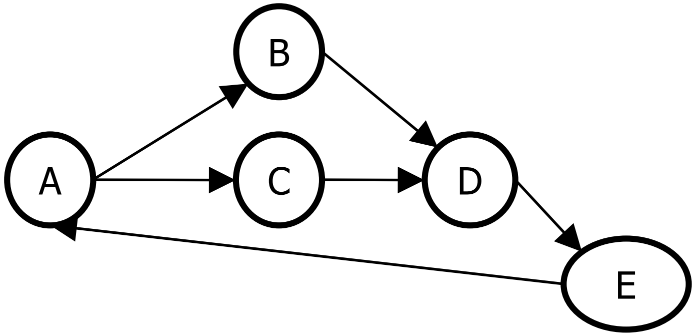
- Why not extend paths by all paths found so-far?
  - And not just edges
  - $M^{2'}=MxM$: Path of length ≤2
    $M^{3'}=M^{2'}xM \cup M^{2'}xM^{2'}$: Path of length ≤2+1 and ≤2+2
    $M^{4'}=M^{3'}xM \cup M^{3'}xM^{2'} \cup M^{3'}xM^{3'}$: …lengths ≤4+1, ≤4+2, ≤4+3/4
    …
- We can save the redundancy
  - $M^{2'}=MxM$: Path of length ≤2
    $M^{4'}=M^{2'}xM^{2'}$: Path of length ≤2+2
    $M^{8'}=M^{4'}xM^{4'}$: …lengths ≤4+4
    …
- Trick: We can stop much earlier
  - The longest shortest path can have length at most n
  - Thus, it suffices to compute $M^{\log(n)'} = \ldots \cup M^{\log(n)'*}xM^{\log(n)'}$

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



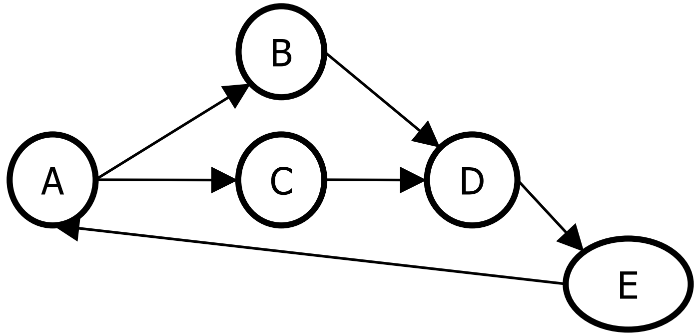
	A	B	C	D	E
A		1	1		
B				1	
C				1	
D					1
E	1				

	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: Connection 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 connections are discovered less often?
  - I.e., less connections are tested

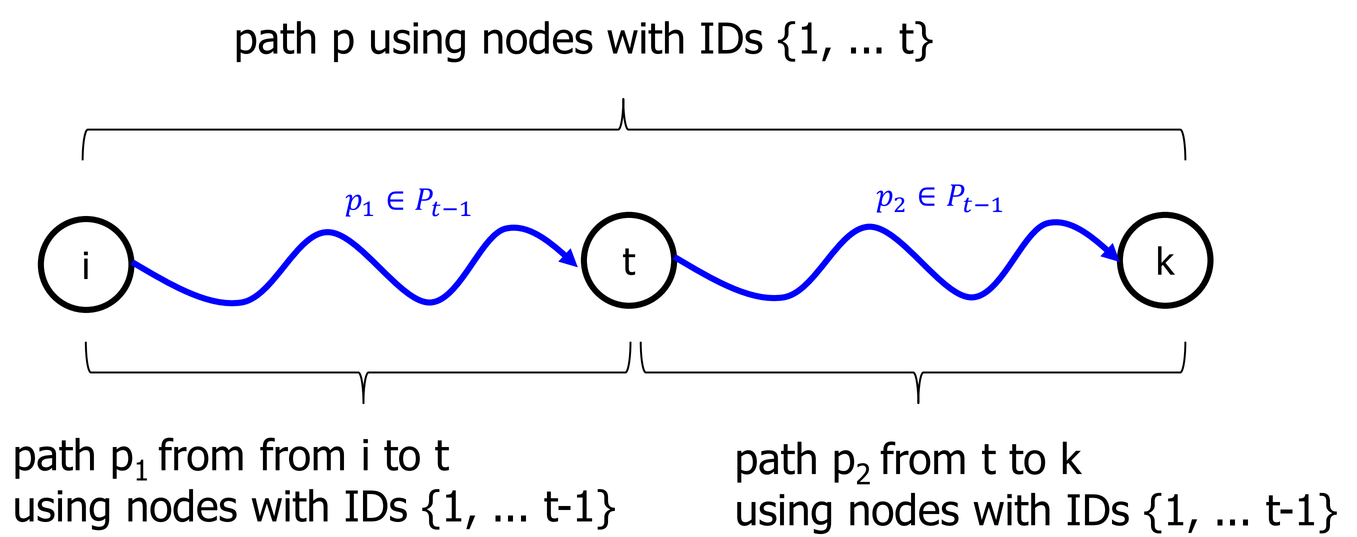
Warshall's Algorithm

Warshall, S. (1962). A theorem on Boolean matrices. *Journal of the ACM 9(1): 11-12.*

- Preparations
  - Fix an arbitrary order of 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
    - Applies to inner nodes of a path, not start and end
  - t gives the highest allowed node ID inside a path
- Idea: Compute P_t inductively
  - We start with P_1
  - 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 $i \xrightarrow{p_1 \in P_{t-1}} t \xrightarrow{p_2 \in P_{t-1}} k$
    - All paths with all IDs <t are already known
    - Node t is the only new player, must be in all new paths
  - We are done once t=n
    - This guarantees correctness – all connections found

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

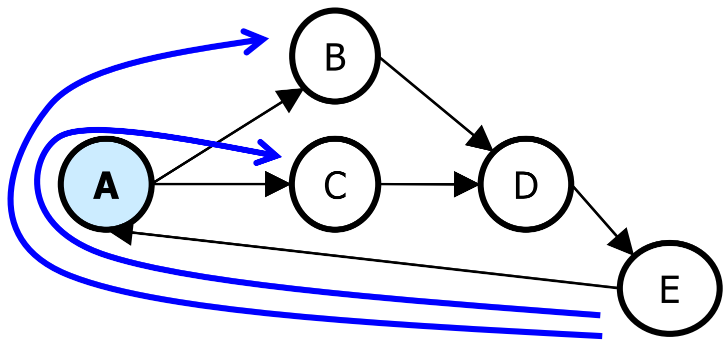


Algorithm

- 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→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				

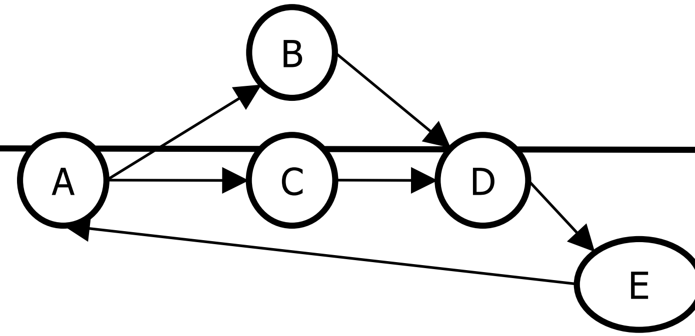
maxID=t=A

	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	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;
```

$O(n^4)$

Drop z-Loop
Swap i and j loop
Rename 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;
```

$O(n^3)$

Content of this Lecture

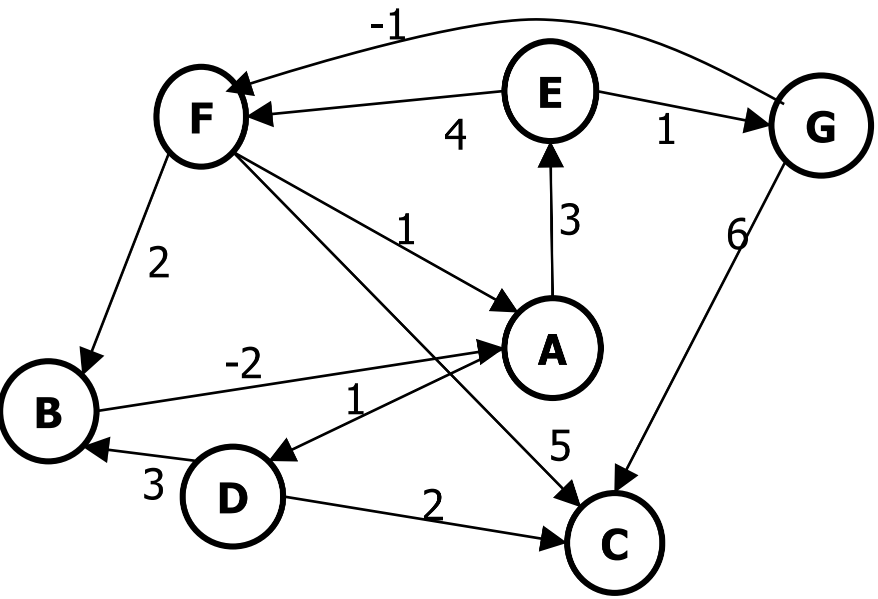
- All-Pairs Shortest Paths
  - Transitive closure: Warshall's algorithm
  - Shortest paths: Floyd's algorithm
- Reachability in Trees

Shortest Paths

Floyd, R. W. (1963). Algorithm 97: Shortest Paths. *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 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	<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	

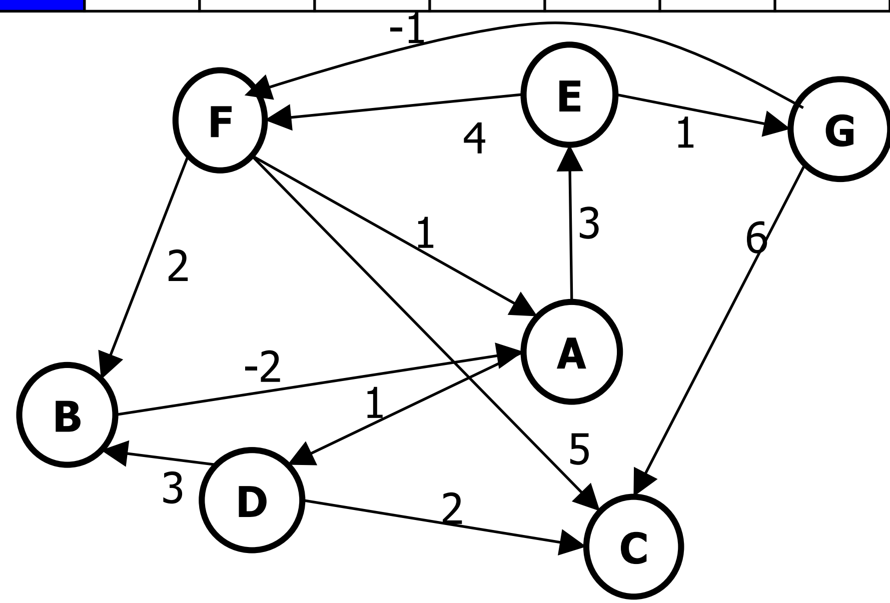
	A	B	C	D	E	F	G
A	2	4	3	1	3	7	4
B	-2	2	1	-1	1	5	2
C							
D	1	3	2	2	4	8	5
E						4	1
F	0	2	3	1	3	7	4
G			6			-1	

Example 3/3

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



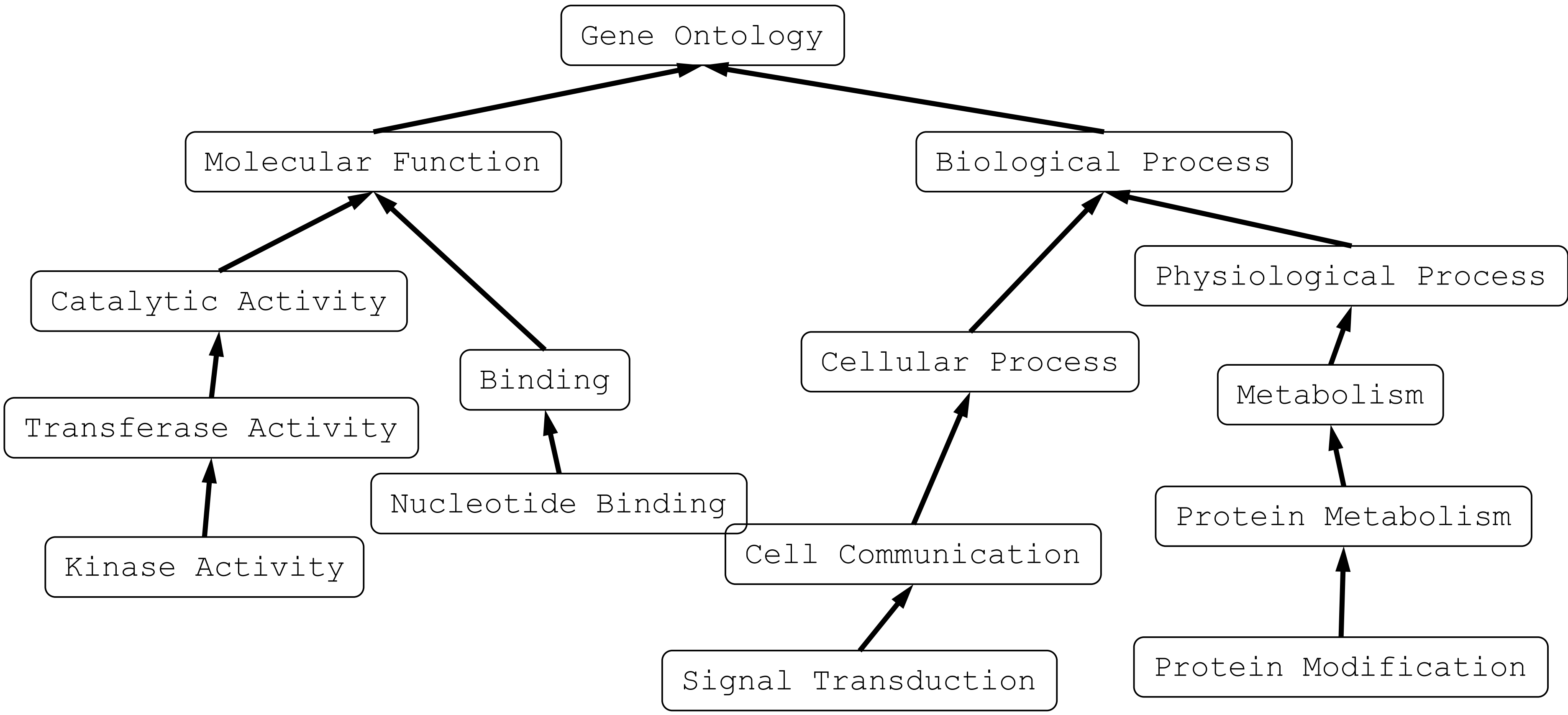
Summary (n=|V|, m=|E|)

- 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 $O(n^2*\log(n)+n*m)$
  - Johnson, Donald B. "Efficient algorithms for shortest paths in sparse networks." *Journal of the ACM (JACM)* 24.1 (1977): 1-13.
  - Faster for sparse graphs
- Storing the results always 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

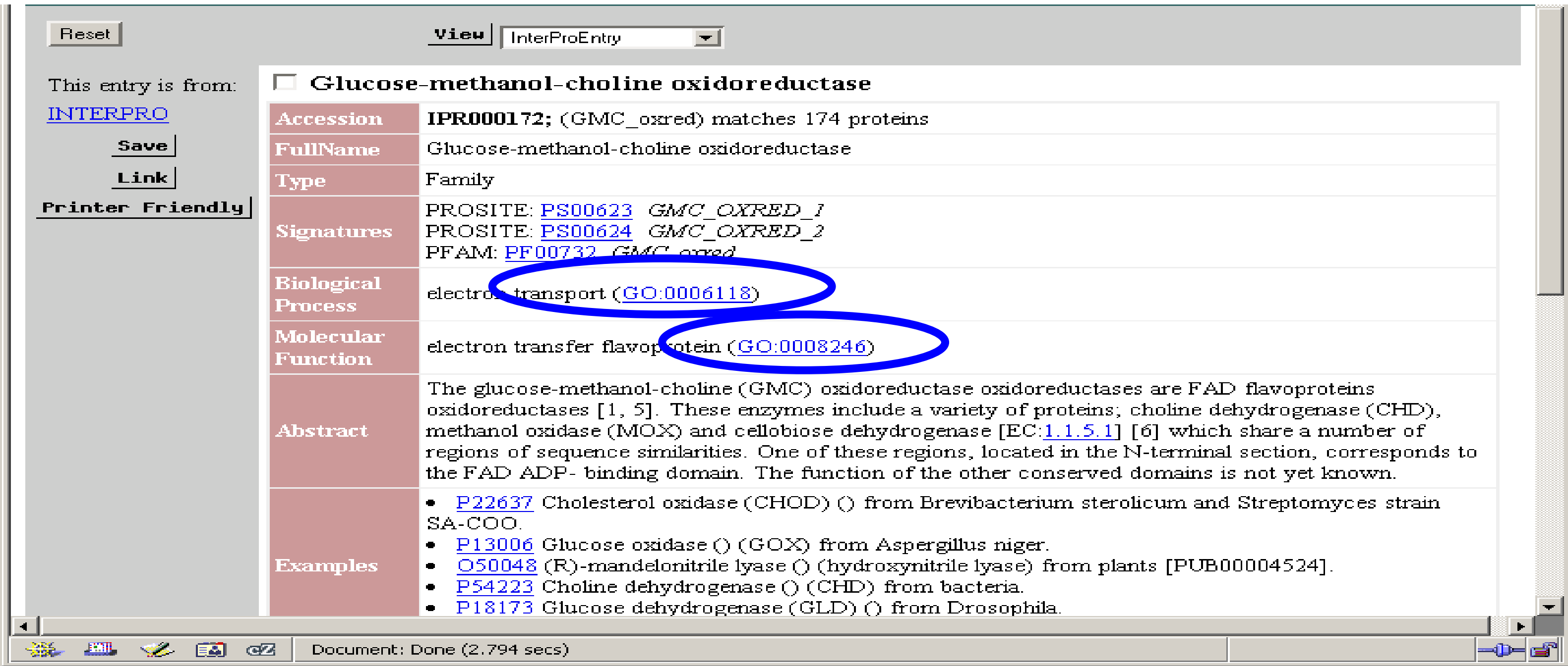
Content of this Lecture

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

Gene Ontology – Describing Gene Function

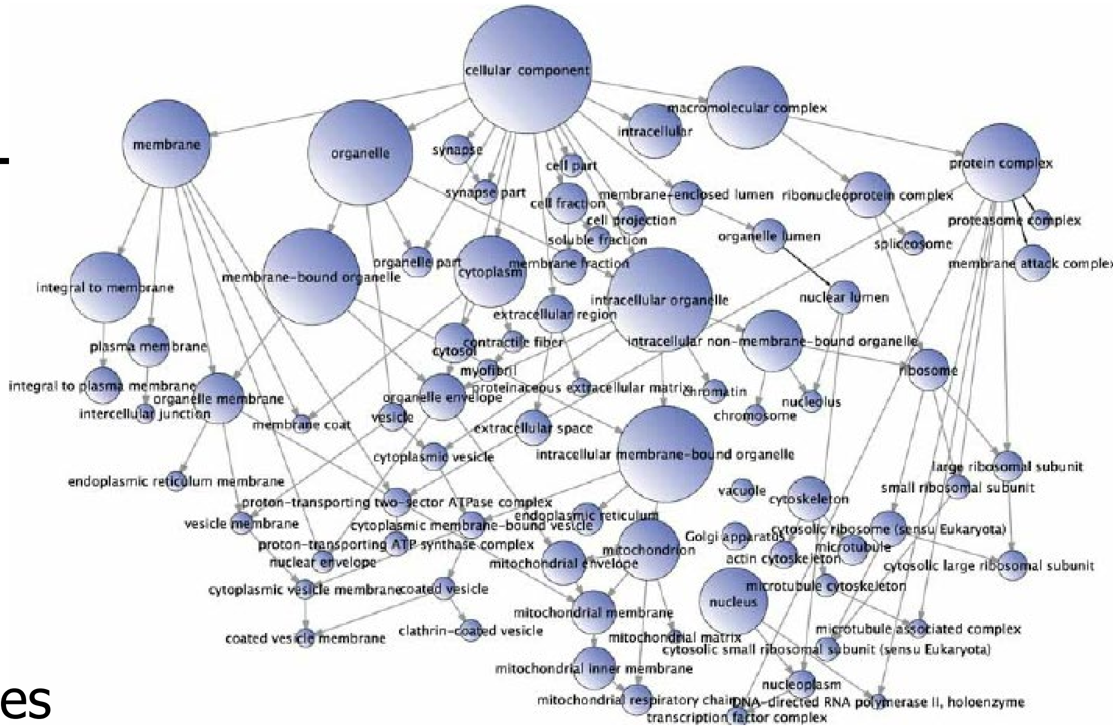


Database Annotation InterPro

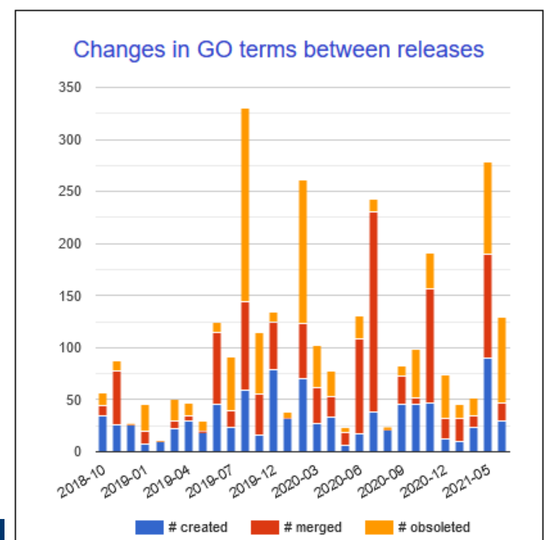


- Used by many databases
- Allows cross-database search
- Provides fixed meaning of terms
  - As informal textual description, not as formal definitions

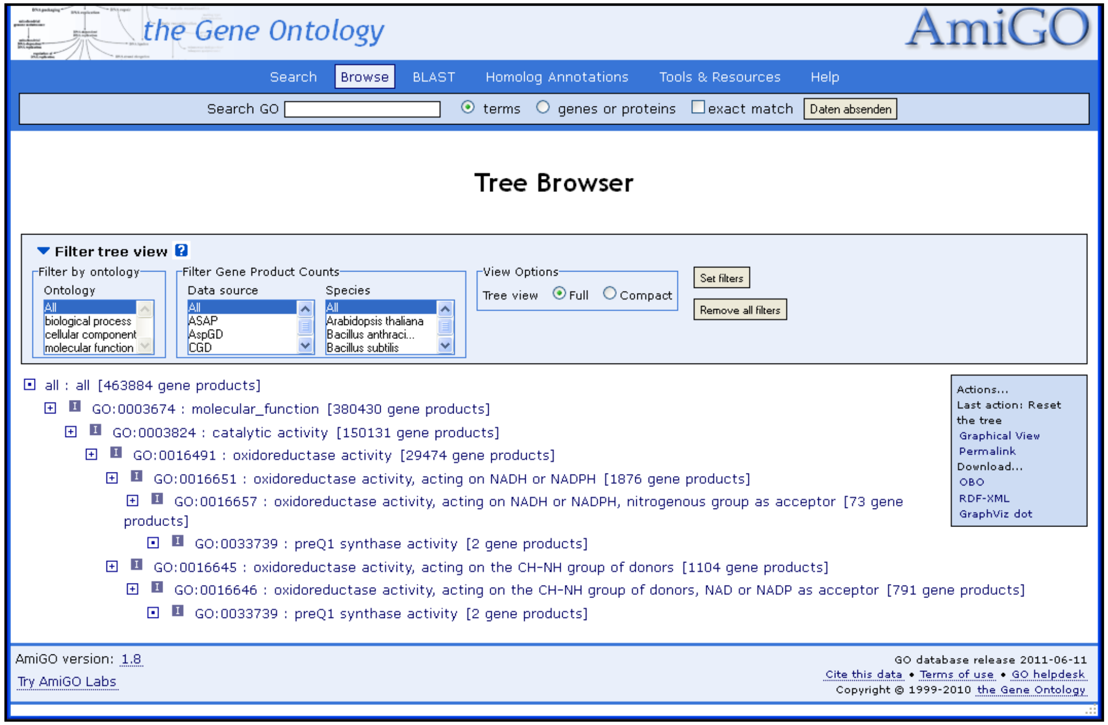
A Large Ontology



- As of 7.7.2021
  - 43917 terms
  - In three subontologies
    - Biological processs
    - Cellular components
    - Molecular functions
  - 3295 obsolete terms
  - Source: http://geneontology.org/stats.html
- Depth: >30



Problem



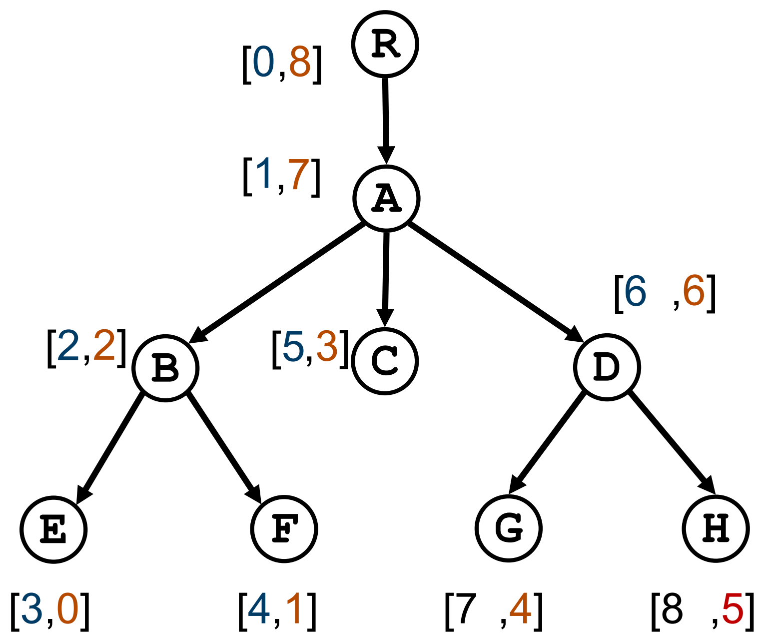
- To see whether a term X IS_A term Y, we need to check whether Y lies on the path from root to X
- Reachability problem

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**



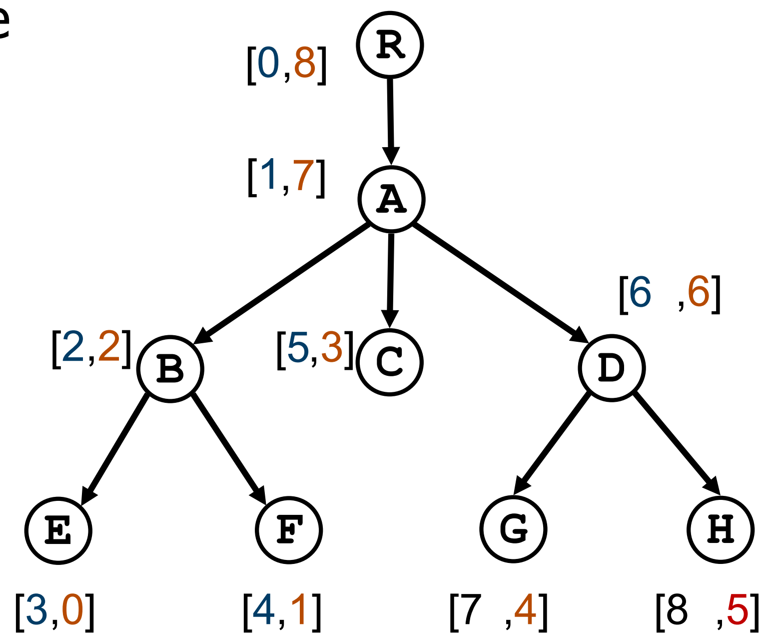
Examples from S. Trissl, 2007

Ancestry and Pre-/Postorder Numbers

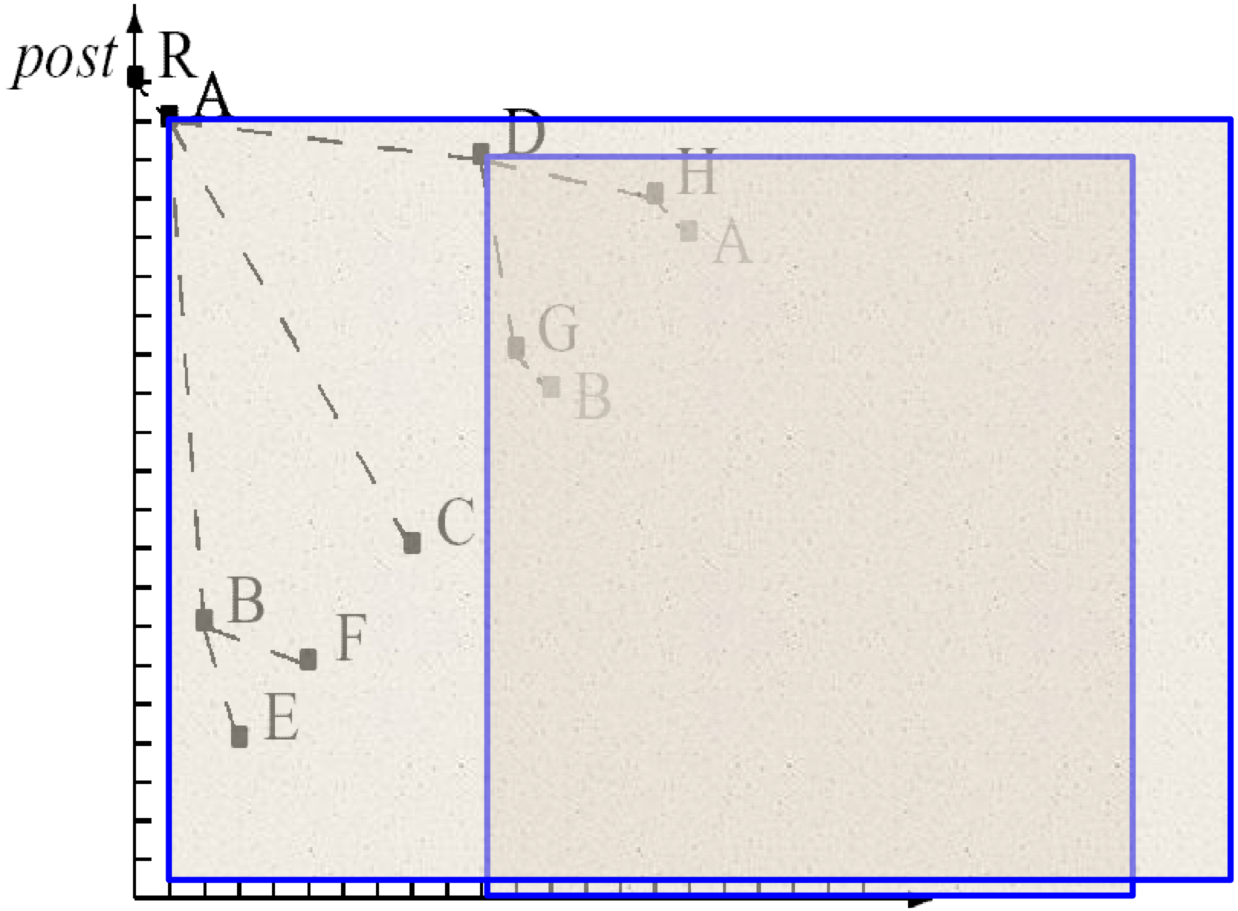
- Trick: A node v is reachable from a node w iff

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

- 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)



Intuition



Exemplary Examination Questions

- We analyzed the Floyd-Warshall algorithm under the assumption that the graph is stored in an adjacency matrix. Repeat the analysis assuming the graph is stored in adjacency lists, one for incoming and one for outgoing lists. First specify which list implementation you want to use (linked list, sorted LL, …)
- Devise an algorithm that find all negative cycles in a weighted digraph and breaks them by removing edges; thus, the final graph must not contain any negative cycle.
- Explain in which sense the Warshall-algorithm implements the dynamic programming paradigm.