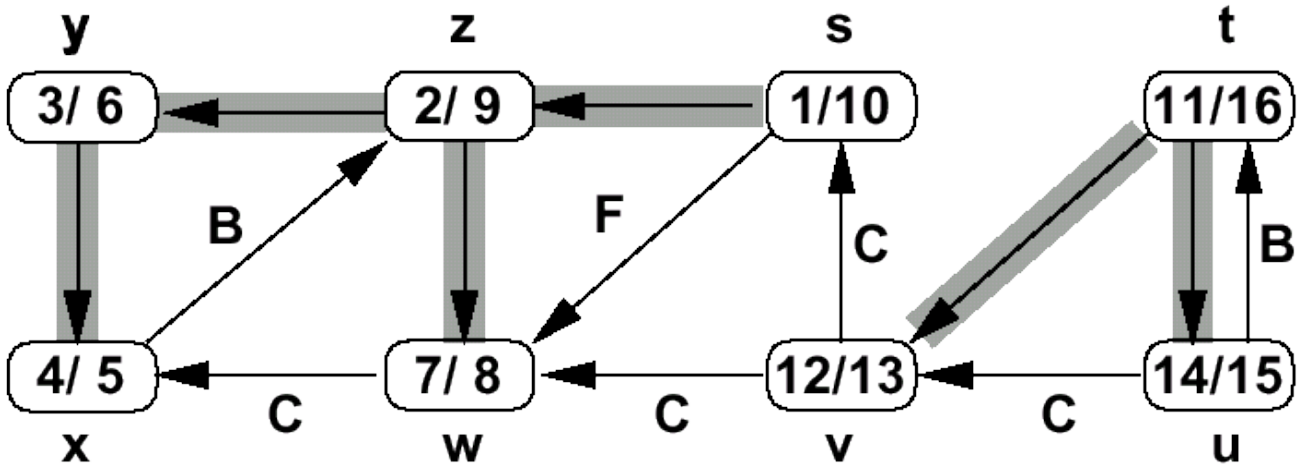
DFS Timestamping

- The DFS algorithm maintains a monotonically increasing global clock
  - discovery time $d[u]$ and finishing time $f[u]$
- For every vertex u, the inequality $d[u] < f[u]$ must hold



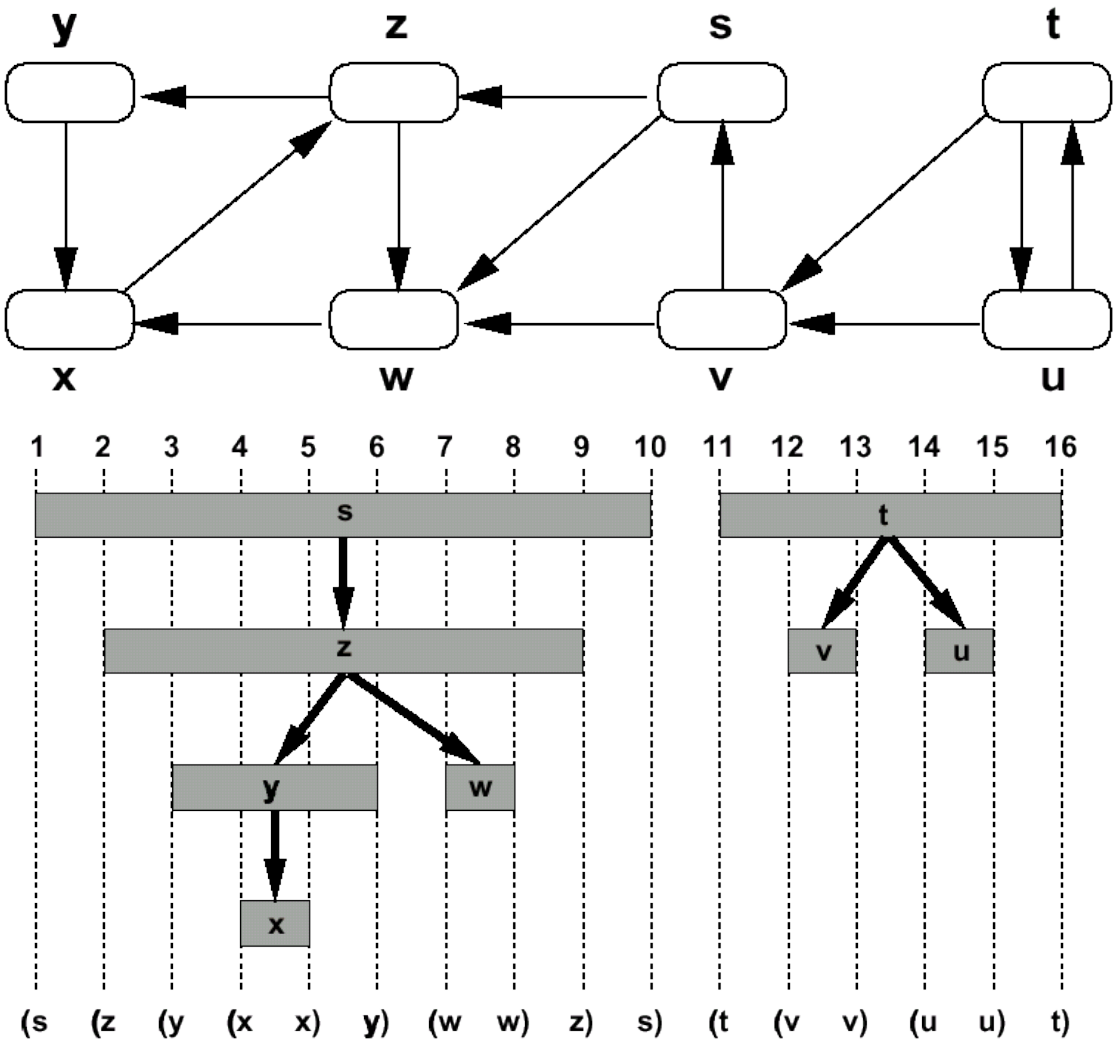
DFS Timestamping

- Vertex u is
  - white before time $d[u]$
  - gray between time $d[u]$ and time $f[u]$, and
  - black thereafter
- Notice the structure througout the algorithm.
  - gray vertices form a linear chain
  - correponds to a stack of vertices that have not been exhaustively explored (DFS-Visit started but not yet finished)

DFS Parenthesis Theorem

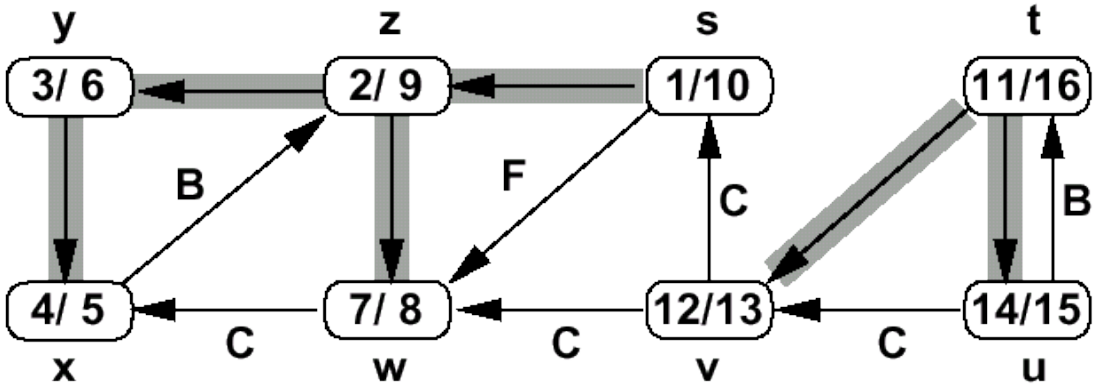
- Discovery and finish times have parenthesis structure
  - represent discovery of *u* with left parenthesis "(u"
  - represent finishin of *u* with right parenthesis "u)"
  - history of discoveries and finishings makes a well-formed expression (parenthesis are properly nested)
- Intuition for proof: any two intervals are either disjoint or enclosed
  - Overlaping intervals would mean finishing ancestor, before finishing descendant or starting descendant without starting ancestor

DFS Parenthesis Theorem (2)



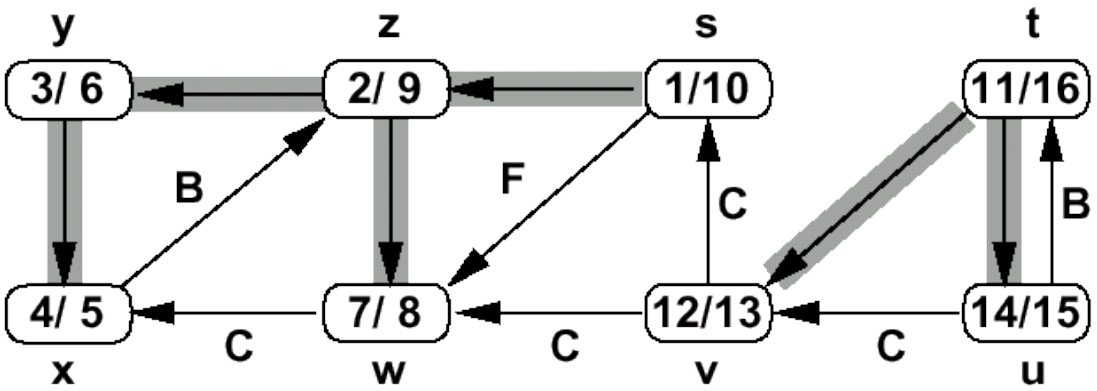
DFS Edge Classification

- Tree edge (gray to white)
  - encounter new vertices (white)
- Back edge (gray to gray)
  - from descendant to ancestor



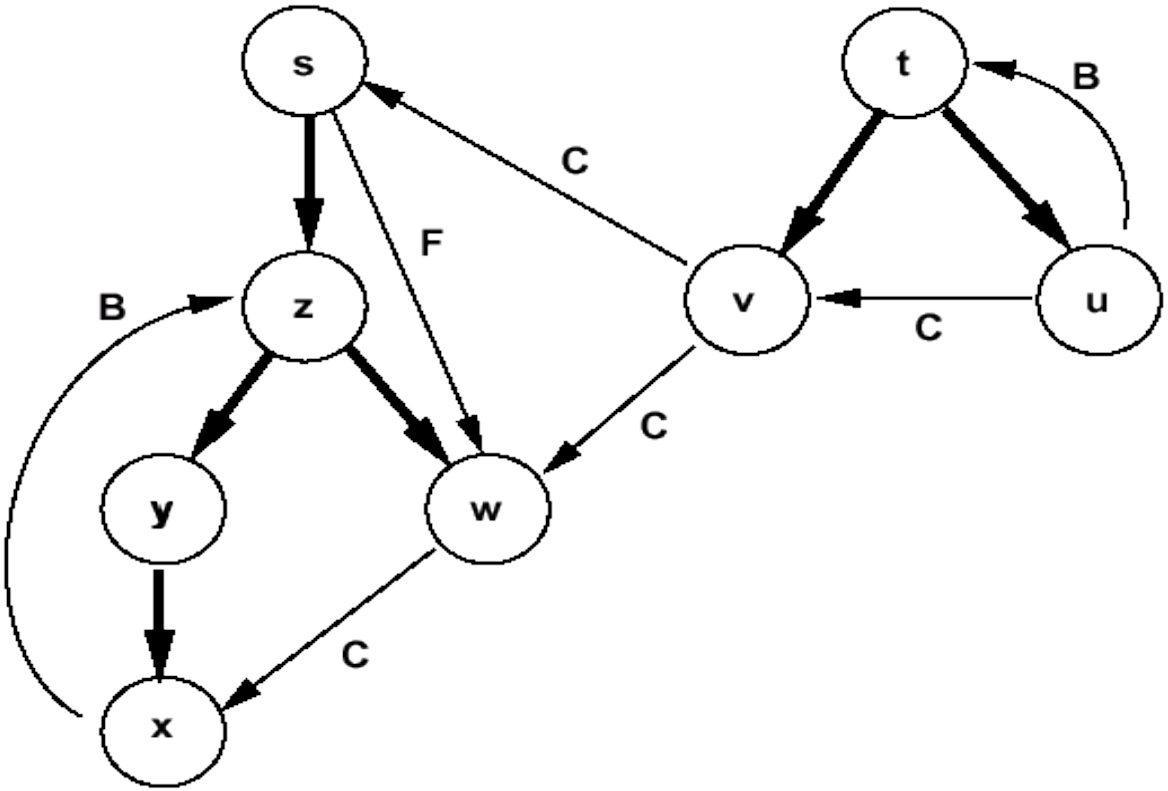
DFS Edge Classification (2)

- Forward edge (gray to black)
  - from ancestor to descendant
- Cross edge (gray to black)
  - remainder – between trees or subtrees



DFS Edge Classification (3)

- Tree and back edges are important
- Most algorithms do not distinguish between forward and cross edges

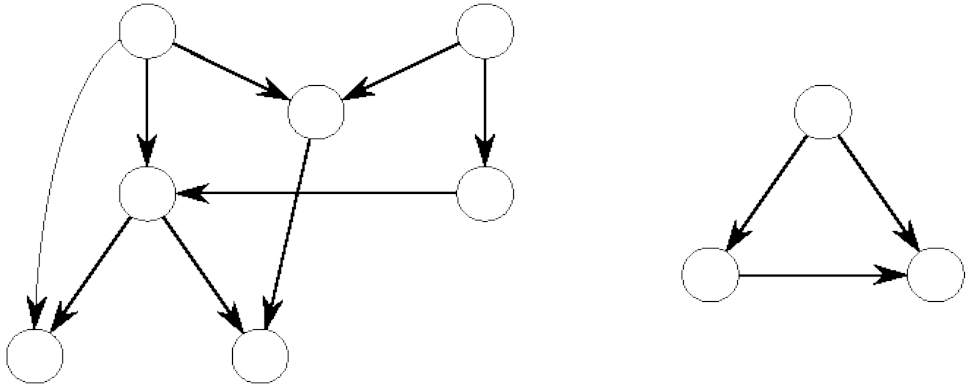


Next:

- Application of DFS: Topological Sort

Directed Acyclic Graphs

- A DAG is a directed graph with no cycles



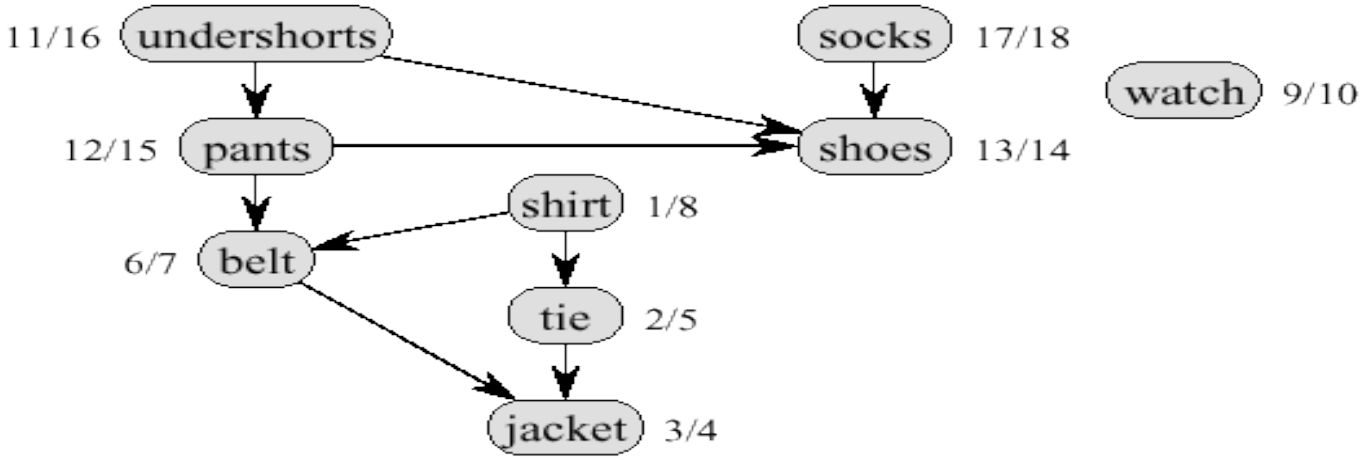
- Often used to indicate precedences among events, i.e., event *a* must happen before *b*
- An example would be a parallel code execution
- Total order can be introduced using **Topological Sorting**

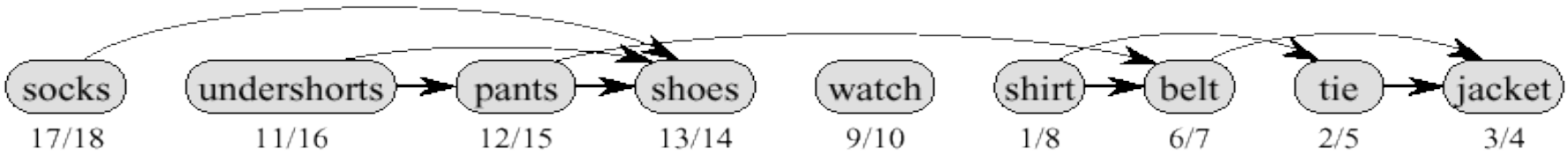
DAG Theorem

- A directed graph *G* is acyclic if and only if a DFS of *G* yields no back edges. Proof:
  - **suppose there is a back edge (*u*,*v*);** *v* is an ancestor of *u* in DFS forest. Thus, there is a path from *v* to *u* in G and (*u*,*v*) completes the cycle
  - **suppose there is a cycle *c*;** let *v* be the first vertex in *c* to be discovered and *u* is a predecessor of *v* in *c*.
    - Upon discovering *v* the whole cycle from *v* to *u* is white
    - We must visit all nodes reachable on this white path before return DFS-Visit(*v*), i.e., vertex *u* becomes a descendant of *v*
    - Thus, (*u,v*) is a back edge
- Thus, we can verify a DAG using DFS!

Topological Sort Example

- Precedence relations: an edge from *x* to *y* means one must be done with *x* before one can do *y*
- Intuition: can schedule task only when all of its subtasks have been scheduled





Topological Sort

- Sorting of a directed acyclic graph (DAG)
- A topological sort of a DAG is a linear ordering of all its vertices such that for any edge (*u*,*v*) in the DAG, *u* appears before *v* in the ordering
- The following algorithm topologically sorts a DAG

Topological-Sort(G)
1) call DFS(G) to compute finishing times *f*[*v*] for each vertex *v*
2) as each vertex is finished, insert it onto the front of a linked list
3) return the linked list of vertices

- The linked lists comprises a total ordering

Topological Sort

- Running time
  - depth-first search: $O(V+E)$ time
  - insert each of the $|V|$ vertices to the front of the linked list: $O(1)$ per insertion
- Thus the total running time is $O(V+E)$

Topological Sort Correctness

- Claim: for a DAG, an edge $(u,v) \in E \Rightarrow f[u] > f[v]$
- When (u,v) explored, u is gray. We can distinguish three cases
  - v = gray
    $\Rightarrow$ (u,v) = back edge (cycle, contradiction)
  - v = white
    $\Rightarrow$ v becomes descendant of u
    $\Rightarrow$ v will be finished before u
    $\Rightarrow$ f[v] < f[u]
  - v = black
    $\Rightarrow$ v is already finished
    $\Rightarrow$ f[v] < f[u]
- The definition of topological sort is satisfied

Next....

Shortest path problems

Single-source shortest paths in weighted graphs

- Shortest-Path Problems
- Properties of Shortest Paths, Relaxation
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Shortest-Paths in DAG's

Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function
  $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest path problems

- Shortest-Path problems
  - Unweighted shortest-paths – BFS.
  - **Single-source, single-destination:** Given two vertices, find a shortest path between them.
  - **Single-source, all destinations:** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.

    [Solution to this problem solves the previous problem efficiently]. Greedy algorithm!
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.

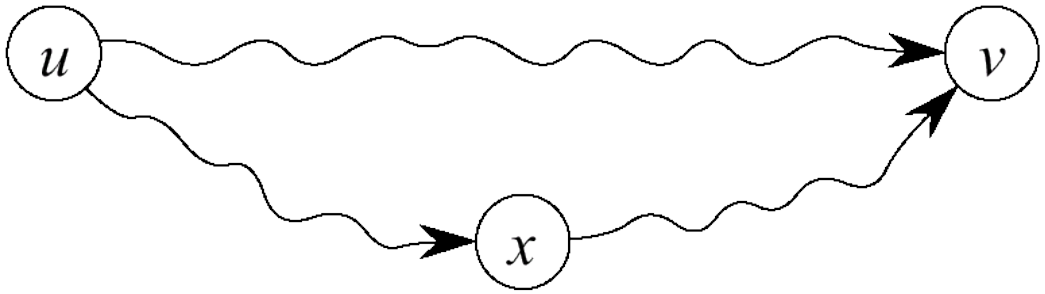
Optimal Substructure

- Theorem: subpaths of shortest paths are shortest paths
- Proof (cut and paste)
  - if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path

Suggests that there may be a greedy algorithm

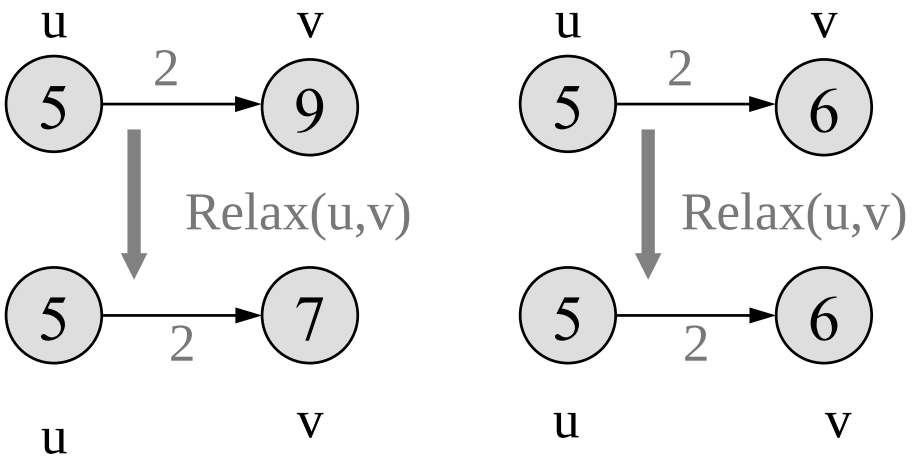
Triangle Inequality

- Definition
  - $\delta(u,v) \equiv$ weight of a shortest path from *u* to *v*
- Theorem
  - $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$ for any *x*
- Proof
  - shortest path u $\in$ v is no longer than any other path u $\in$ v – in particular, the path concatenating the shortest path u $\in$ x with the shortest path x $\in$ v



Relaxation

- For each vertex in the graph, we maintain d[v], the estimate of the shortest path from *s*, initialized to ∞ at start
- Relaxing an edge (u,v) means testing whether we can improve the shortest path to v found so far by going through u




```
Relax (u,v,w)
if d[v] >
   d[u]+w(u,v)then
    d[v] ← d[u]+w(u,v)
    π[v] ← u
```

Dijkstra's Algorithm

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use *Q*, priority queue keyed by *d*[*v*] (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some *d* decreases)
- Basic idea
  - maintain a set *S* of solved vertices
  - at each step select "closest" vertex *u*, add it to *S*, and relax all edges from *u*

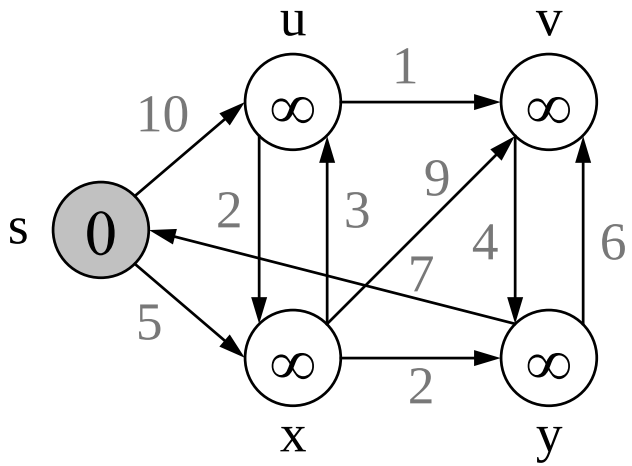
Dijkstra's Algorithm: pseudocode

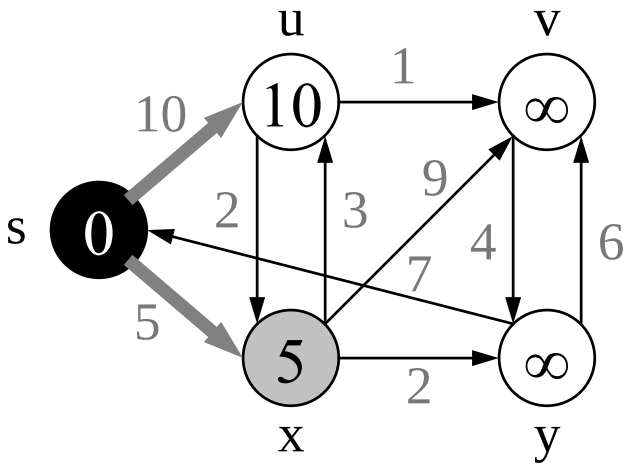
- Graph G, weight function w, root s

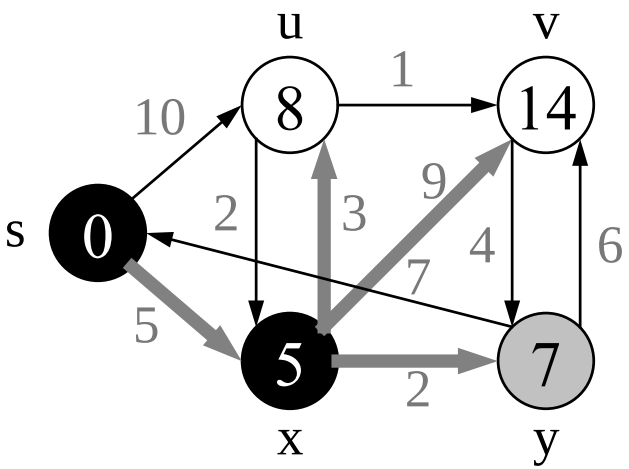
```
DIJKSTRA(G, w, s)
 1 for each v ∈ V
 2       do d[v] ← ∞
 3 d[s] ← 0
 4 S ← ∅   ▷ Set of discovered nodes
 5 Q ← V
 6 while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          S ← S ∪ {u}
 9          for each v ∈ Adj[u]
10              do if d[v] > d[u] + w(u, v)
11                    then d[v] ← d[u] + w(u, v)
```

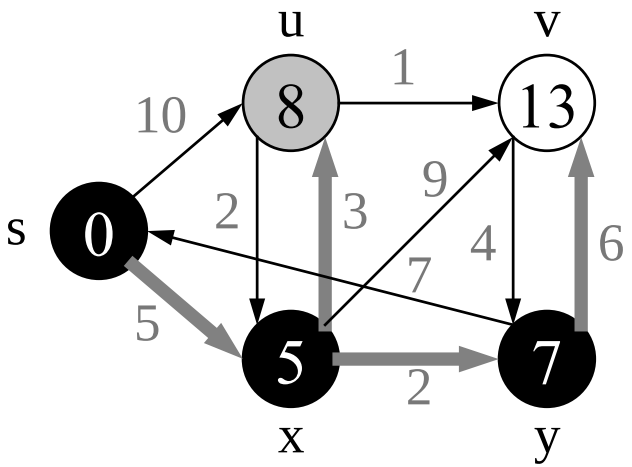
relaxing edges

Dijkstra's Algorithm: example

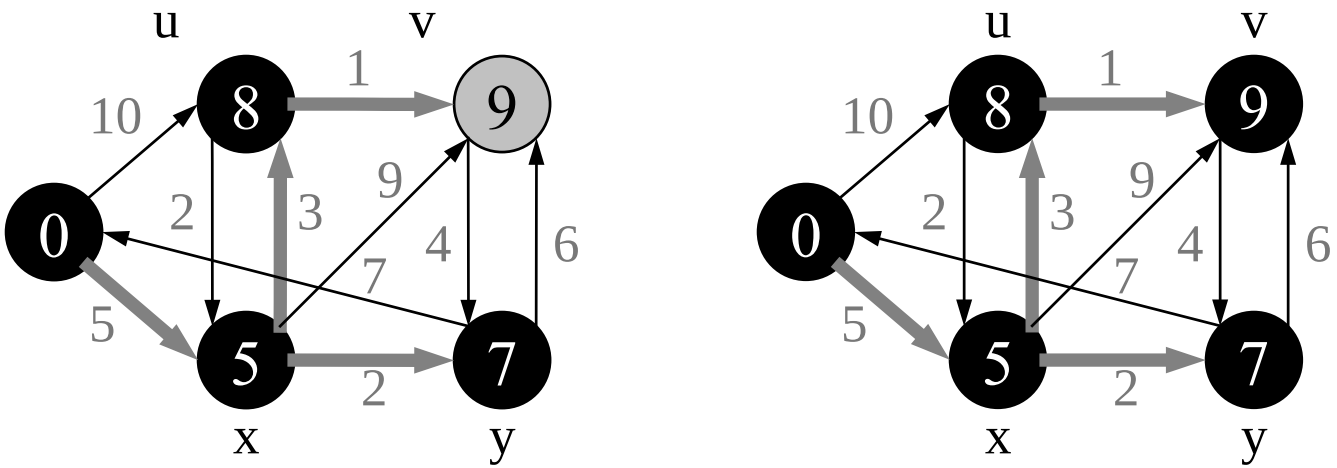








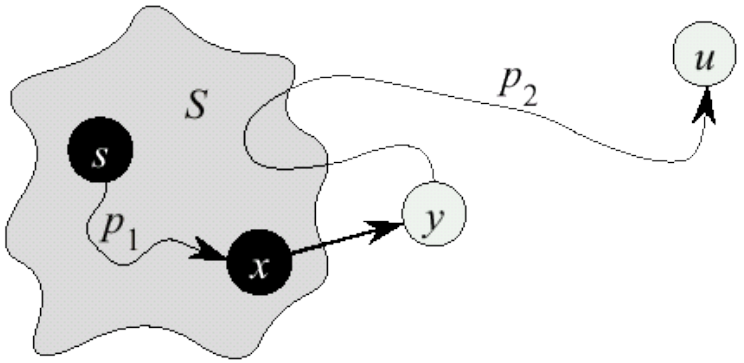
Dijkstra's Algorithm: example (2)



- Observe
  - relaxation step (lines 10-11)
  - setting $d[v]$ updates Q (needs Decrease-Key)
  - similar to Prim's MST algorithm

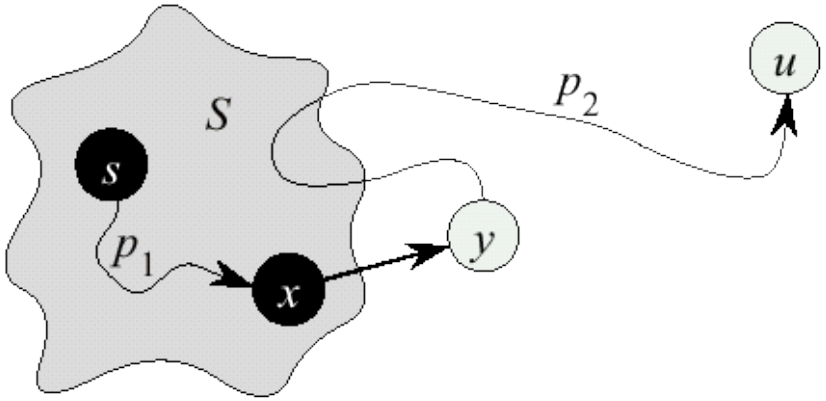
Dijkstra's Algorithm: correctness

- We will prove that **whenever *u* is added to *S***, $d[u] = d(s,u)$, i.e., that d is minimum, and that equality is maintained thereafter
- Proof
  - Note that $\forall v$, $d[v] \geq d(s,v)$
  - Let u be the first **vertex picked** such that there is a shorter path than $d[u]$, i.e., that $\Rightarrow d[u] > d(s,u)$
  - We will show that this assumption leads to a contradiction



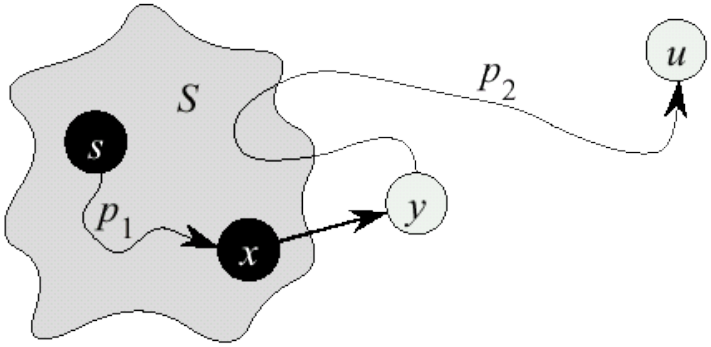
Dijkstra's Algorithm: correctness (2)

- Let y be the first vertex $\in V - S$ on the actual shortest path from s to u, then it must be that $d[y] = \delta(s,y)$ because
  - $d[x]$ is set correctly for y's predecessor $x \in S$ on the shortest path (by choice of u as the first vertex for which d is set incorrectly)
  - when the algorithm inserted x into S, it relaxed the edge (x,y), assigning $d[y]$ the correct value



Dijkstra's Algorithm: correctness (3)

$$
\begin{aligned}
d[u] &> \delta(s,u) && \text{(initial assumption)} \\
&= \delta(s,y)+\delta(y,u) && \text{(optimal substructure)} \\
&= d[y]+\delta(y,u) && \text{(correctness of } d[y]\text{)} \\
&\geq d[y] && \text{(no negative weights)}
\end{aligned}
$$



- But $d[u] > d[y] \Rightarrow$ algorithm would have chosen y (from the PQ) to process next, not u $\Rightarrow$ Contradiction
- Thus d[u] = $\delta(s,u)$ at time of insertion of u into S, and Dijkstra's algorithm is correct

Dijkstra's Algorithm: running time

- Extract-Min executed $|V|$ time
- Decrease-Key executed $|E|$ time
- Time = $|V| T_{\text{Extract-Min}} + |E| T_{\text{Decrease-Key}}$
- T depends on different Q implementations

Q	T(Extract-Min)	T(Decrease-Key)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ (amort.)	$O(V \lg V + E)$

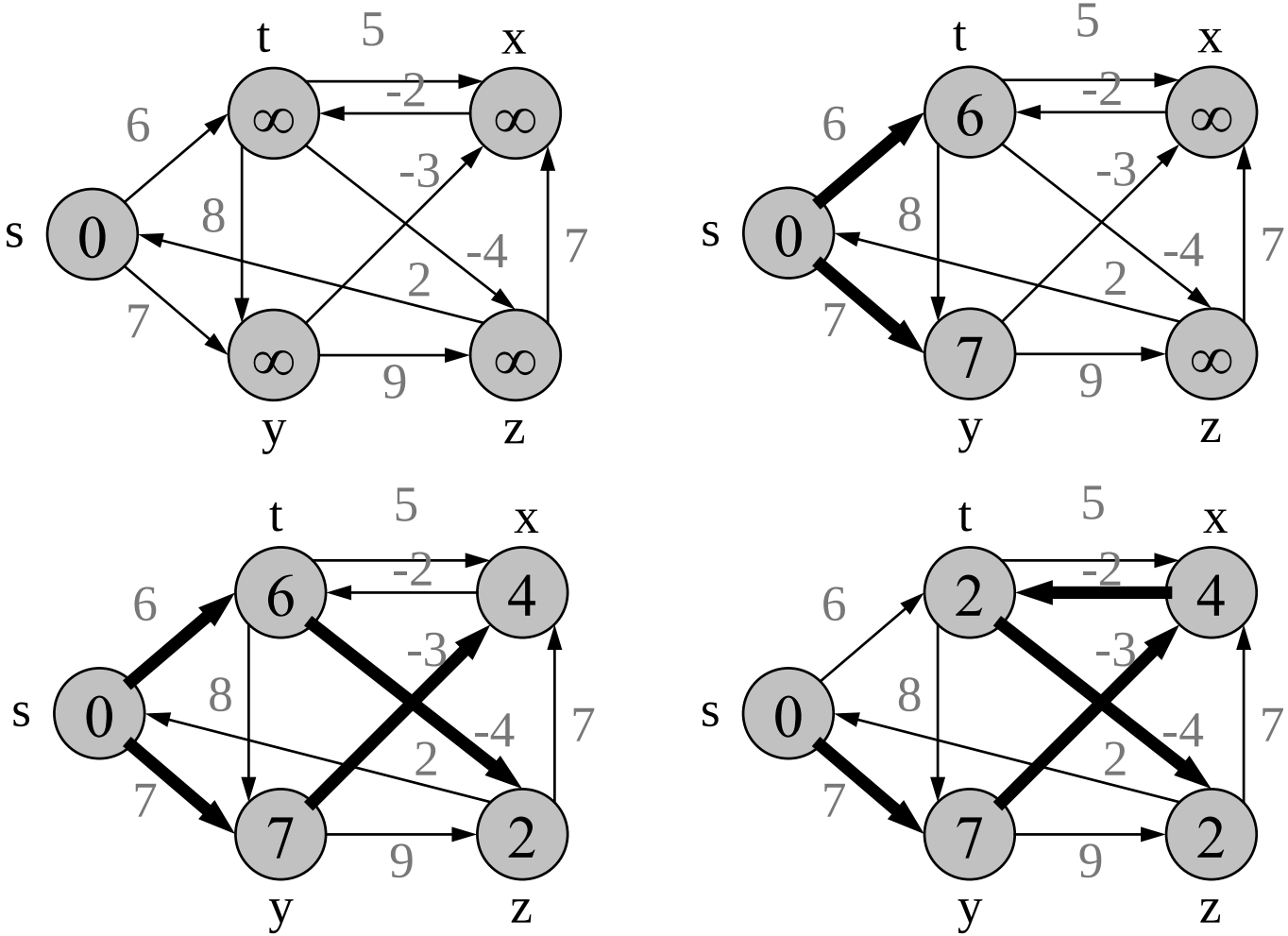
Bellman-Ford Algorithm

- Dijkstra's doesn't work when there are negative edges:
  - Intuition: we can not be greedy any more on the assumption that the lengths of paths will only increase in the future
- Bellman-Ford algorithm detects negative cycles (returns *false*) or returns the shortest path-tree

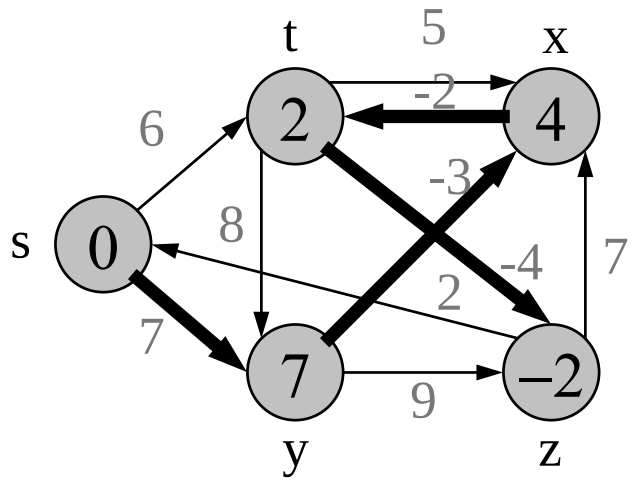
Bellman-Ford Algorithm

```
Bellman-Ford(G,w,s)
01 for each v ∈ V[G]
02    d[v] ← ∞
03 d[s] ← 0
04 π[s] ← NIL
05 for i ← 1 to |V[G]|-1 do
06     for each edge (u,v) ∈ E[G] do
07         Relax (u,v,w)
08 for each edge (u,v) ∈ E[G] do
09     if d[v] > d[u] + w(u,v) then return false
10 return true
```

Bellman-Ford Algorithm: example



Bellman-Ford Algorithm: example (2)



- Bellman-Ford running time:
  - $(|V|-1)|E| + |E| = \Theta(|V||E|)$

Bellman-Ford Algorithm: correctness

- Let $\delta_i(s,u)$ denote the length of path from s to *u,* that is shortest among all paths, that contain at most *i* edges
- Prove by induction that $d[u] = \delta_i(s,u)$ after the *i*-th iteration of Bellman-Ford
  - Base case (*i*=0) trivial
  - Inductive step (say $d[u] = \delta_{i-1}(s,u)$):
    - Either $\delta_i(s,u) = \delta_{i-1}(s,u)$
    - Or $\delta_i(s,u) = \delta_{i-1}(s,z) + w(z,u)$
    - In an iteration we try to relax each edge (*(z,u)* also), so we will catch both cases, thus $d[u] = \delta_i(s,u)$

Bellman-Ford Algorithm: correctness (2)

- After *n-1* iterations, $d[u] = \delta_{n-1}(s,u)$, for each vertex *u.*
- If there is still some edge to relax in the graph, then there is a vertex *u*, such that

  $\delta_n(s,u) < \delta_{n-1}(s,u)$. But there are only *n* vertices in *G* – we have a cycle, and it must be negative.
- Otherwise, $d[u] = \delta_{n-1}(s,u) = \delta(s,u)$, for all *u,* since any shortest path will have at most *n-1* edges

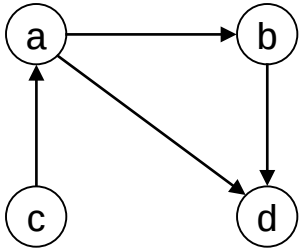
Next....

Next: All-pairs shortest paths in weighted graphs

- Matrix multiplication and shortest-paths
- Floyd Warshall algorithm
- Transitive closure

All-pairs shortest paths

- Suppose that we want to calculate information about shortest paths between all pairs of vertices.



- We have a matrix W of weights:

$$\begin{pmatrix} 0 & 1 & \infty & 1 \\ \infty & 0 & \infty & 1 \\ 1 & 0 & 0 & 0 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

- We want a matrix:

$$\begin{pmatrix} 0 & 1 & \infty & 1 \\ \infty & 0 & \infty & 1 \\ 1 & 2 & 0 & 2 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

A Recursive Solution

- $l_{ij}^{(0)} = 0$ if $i=j$

  $= \infty$ otherwise
- $l_{ij}^{(m)} = \min (l_{ij}^{(m-1)}, \min_{1 \le k \le n} \{l_{ik}^{(m-1)} + w_{kj}\})$

  $= \min_{1 \le k \le n} \{l_{ik}^{(m-1)} + w_{kj}\}$

$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} \ldots\ldots$

Matrix multiplication:

- If A is the adjacency matrix for a graph G, then the *ij* th entry of A^n is exactly the number of ways you can get from vertex i to vertex j in exactly n steps.

$$\left(A^{m+1}\right)_{i\ j} = \sum_{k=1}^{q} (A^m)_{i\ k} A_k$$

ways to get from i to k in exactly m steps

ways to get from k to j in one step

If we replace addition of elements by *minimum*, and multiplication of elements by *addition*, then the *ij* th entry of W^n is exactly the shortest path from vertex i to vertex j in at most n steps.

$$\left(W^{m+1}\right)_{i\ j} = \min_{k=1}^{q} \left((W^m)_{i\ k} + W_k\right)_j$$

Shortest path weight for m steps from i to k

Weight for a further step from k to j

Matrix Multiplication contd.

- As in Bellman-Ford, no shortest path has more than |V|-1 vertices in it. Therefore, all the information that we need can be read from the entries in $W^{|V|-1}$.

- Each matrix "multiplication" takes $O(V^3)$.

Matrix Multiplication - complexity

- Calculating $W^{|V|-1}$ takes:
  - $O(V^4)$ if we do naïve exponentiation:
    - $A^0 = I$
    - $A^{m+1} = A\,A^m$
  - Q: How many multiplications are required to compute x^n ?
  - $O(V^3 \log V)$ if we do fast exponentiation:
    - $A^0 = I$
    - $A^1 = A$
    - $A^{2m} = (A^m)^2$
    - $A^{2m+1} = A\,(A^m)^2$

The Floyd-Warshall algorithm

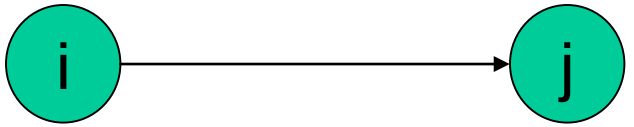
- Instead of increasing the length of the path allowed at each step, suppose that we increase the number of vertices that can be used in forming such paths.

- Let $D^{(k)}$ be the matrix whose *ij* th component is the shortest-path weight for a path from vertex i to vertex j using only vertices 1 though k as intermediates.

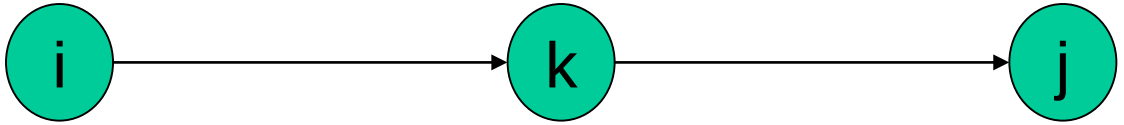
- Note that $D^{(0)} = W$. How can we calculate $D^{(n+1)}$ in terms of $D^{(n)}$?

Floyd-Warshall algorithm – contd.

- A shortest path from i to j with intermediate vertices in 1..k is either:
  - A shortest path from i to j with intermediate vertices in 1..(k-1).



  - A shortest path from i to k, and a shortest path from k to j, both with vertices in 1..(k-1).



- Hence, for k>1, we can define:

$$d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$$

The Floyd-Warshall algorithm

- Let n = |V|, and calculate all F[k] values using:

Time *and space* complexity are $O(V^3)$

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4        do for i ← 1 to n
5              do for j ← 1 to n
6                    do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

Floyd-Warshall algorithm - improvement

- In fact, we can do better - we only want $D^{(n)}$:
- Store only $D^{(n)}$
- Time complexity is $O(V^3)$, space complexity is $O(V^2)$.

Transitive closure

Given a directed graph G = (V,E), construct a new graph G' = (V,E') in which (i,j) $\in$ E' if there is a path From i to j in G.

- $t_{ij}^{(0)} = 0$ if $i \neq j$ and $(i,j) \notin E$

  $= 1$ if $i=j$ or $(i,j) \in E$

And for m>0

$$t_{ij}^{(m)} = t_{ij}^{(m-1)} \vee (t_{im}^{(m-1)} \wedge t_{mj}^{(m-1)})$$

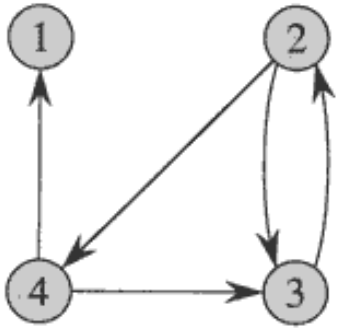
- Reachability queries

Transitive closure algorithm

Very similar to Floyd Warshall:

```
TRANSITIVE-CLOSURE(G)
 1  n ← |V[G]|
 2  for i ← 1 to n
 3       do for j ← 1 to n
 4            do if i = j or (i, j) ∈ E[G]
 5                 then t_ij^(0) ← 1
 6                 else t_ij^(0) ← 0
 7  for k ← 1 to n
 8       do for i ← 1 to n
 9            do for j ← 1 to n
10                 do t_ij^(k) ← t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
11  return T^(n)
```

Transitive closure example



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 25.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

Summary

- We have seen different algorithms for:
  - computing spanning trees;
  - computing minimum spanning trees;
  - computing single-source shortest paths;
  - computing all-pairs shortest paths.
  - Computing transitive closure.
- Greedy algorithms and dynamic programming play key roles in these algorithms.