CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta@cs.yorku.ca) ext 77875

Lectures: Tues (BC 215), 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

Textbook: Cormen, Leiserson, Rivest, Stein.
Introduction to Algorithms (3rd Edition)

Read on your own

- Strongly connected components

(22.3 in Edition 2).

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

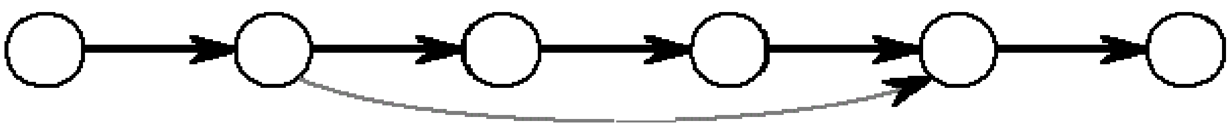
Shortest path problems

- 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

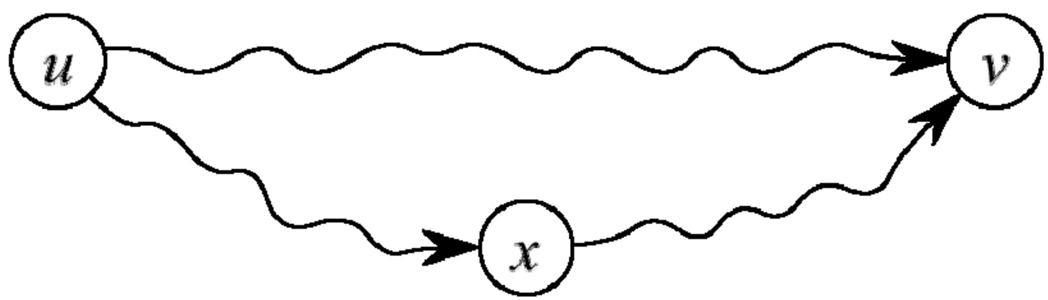
- Theorem: subpaths of shortest paths are shortest paths
- Proof (cut and paste)
  - – if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path



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 ∈ v is no longer than any other path u ∈ v – in particular, the path concatenating the shortest path u ∈ x with the shortest path x ∈ v

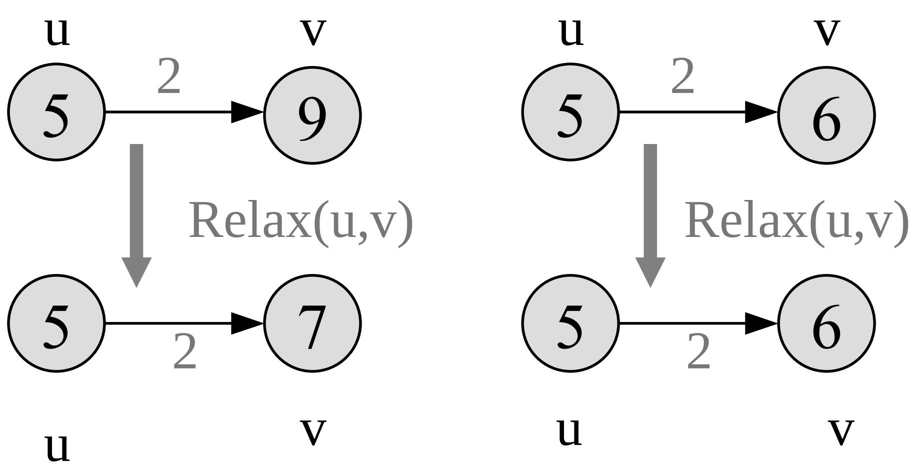


Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small "lengths" would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)
  - Any shortest-path in graph *G* can be no longer than *n* – 1 edges, where *n* is the number of vertices

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*

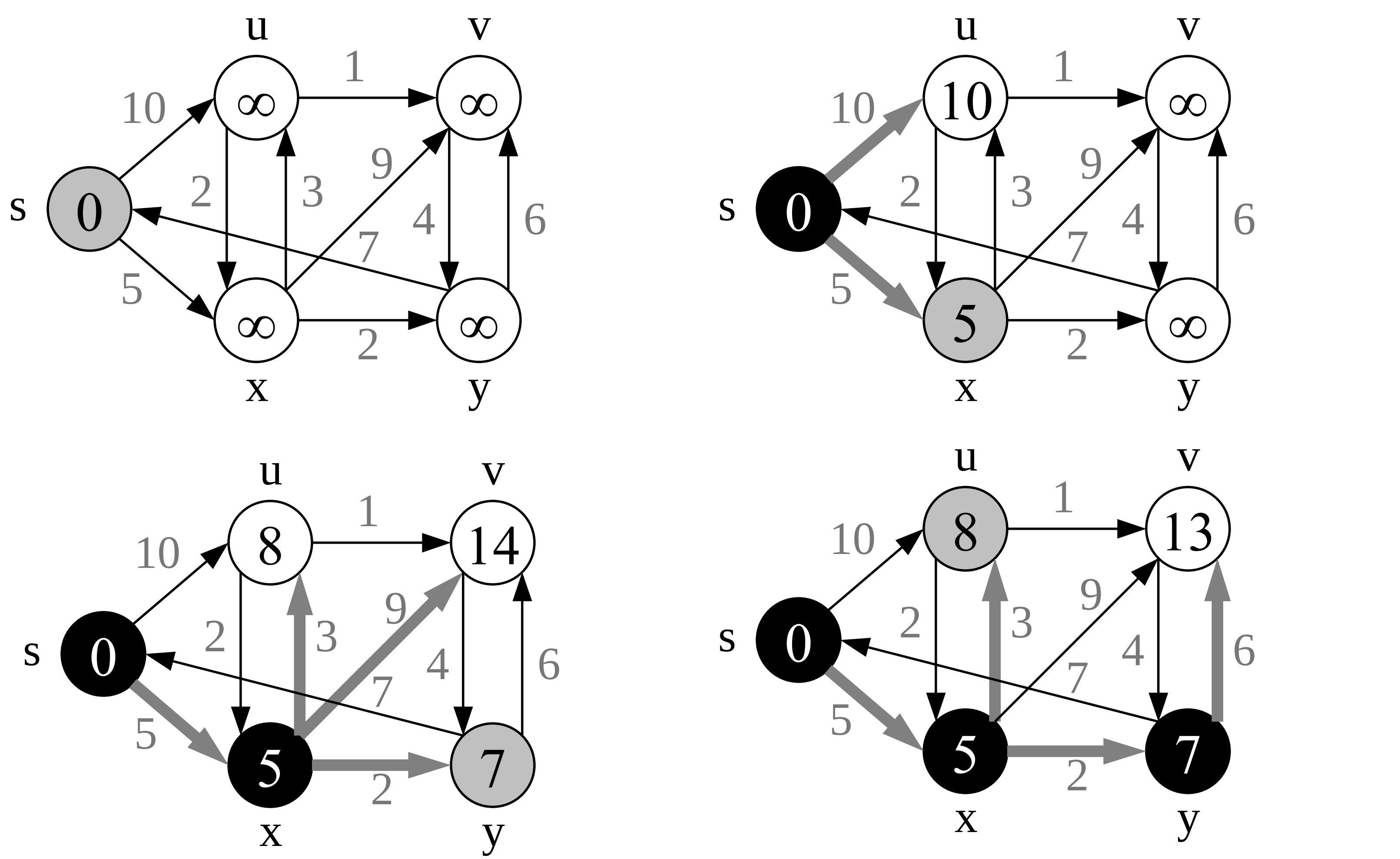
Dijkstra's Algorithm: pseudocode

- Graph *G*, weight function *w*, root *s*

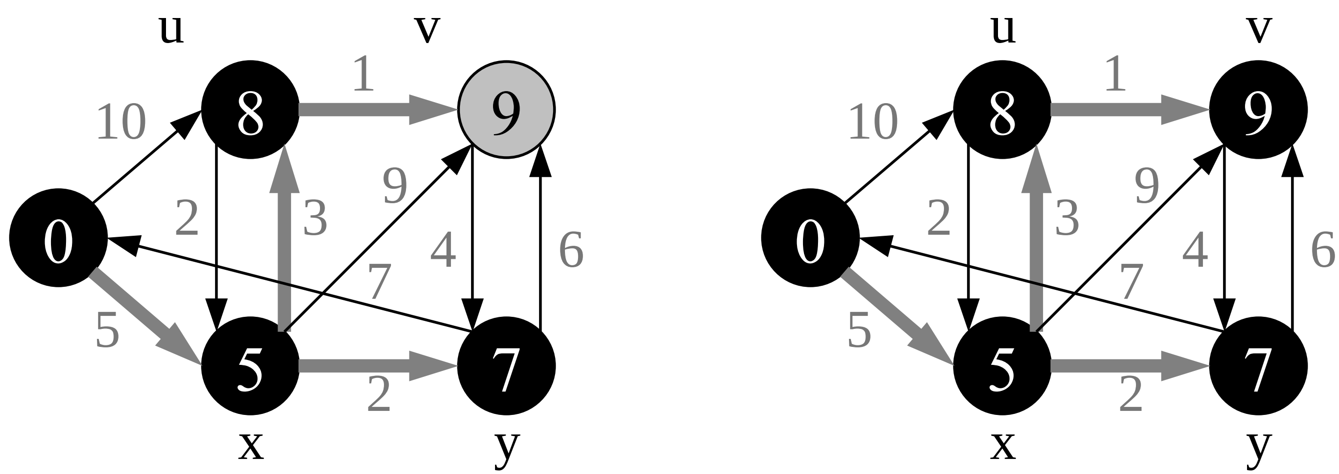
```
DIJKSTRA(G, w, s)
for each v ∈ V
        do d[v] ← ∞
d[s] ← 0
S ← ∅    ▷ Set of discovered nodes
Q ← V
while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
            S ← S ∪ {u}
            for each v ∈ Adj[u]
                do if d[v] > d[u] + w(u, v)
                        then d[v] ← d[u] + w(u, v)
```

relaxing edges

Dijkstra's Algorithm: example



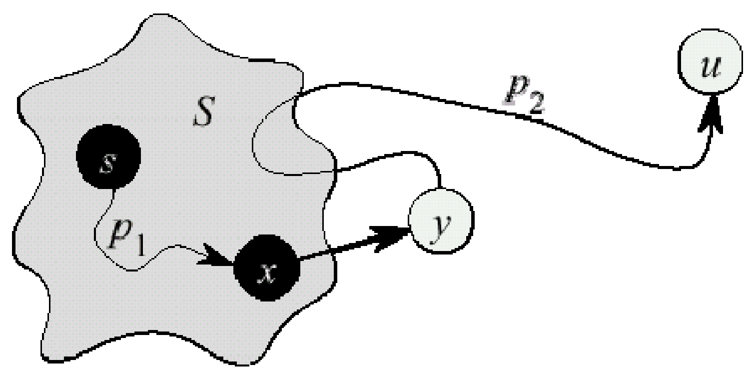
Dijkstra's Algorithm: example (2)



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

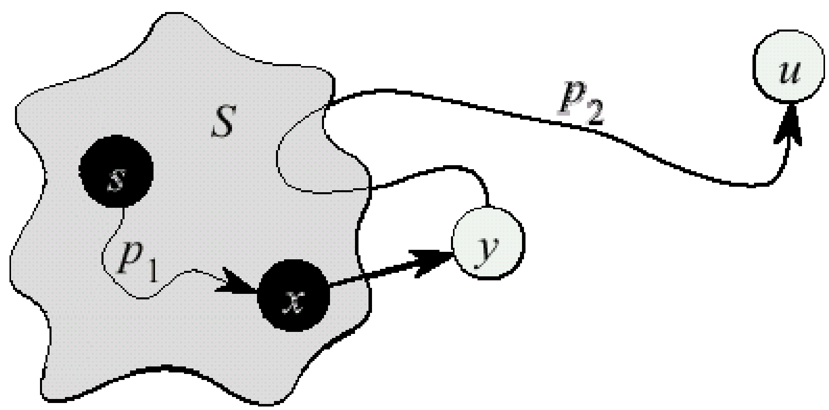
Dijkstra's Algorithm: correctness

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



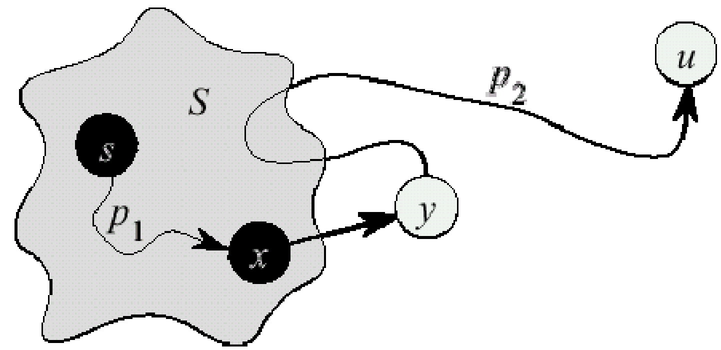
Dijkstra's Algorithm: correctness (2)

- 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)

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



- 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.)	$O(V \lg V + E)$

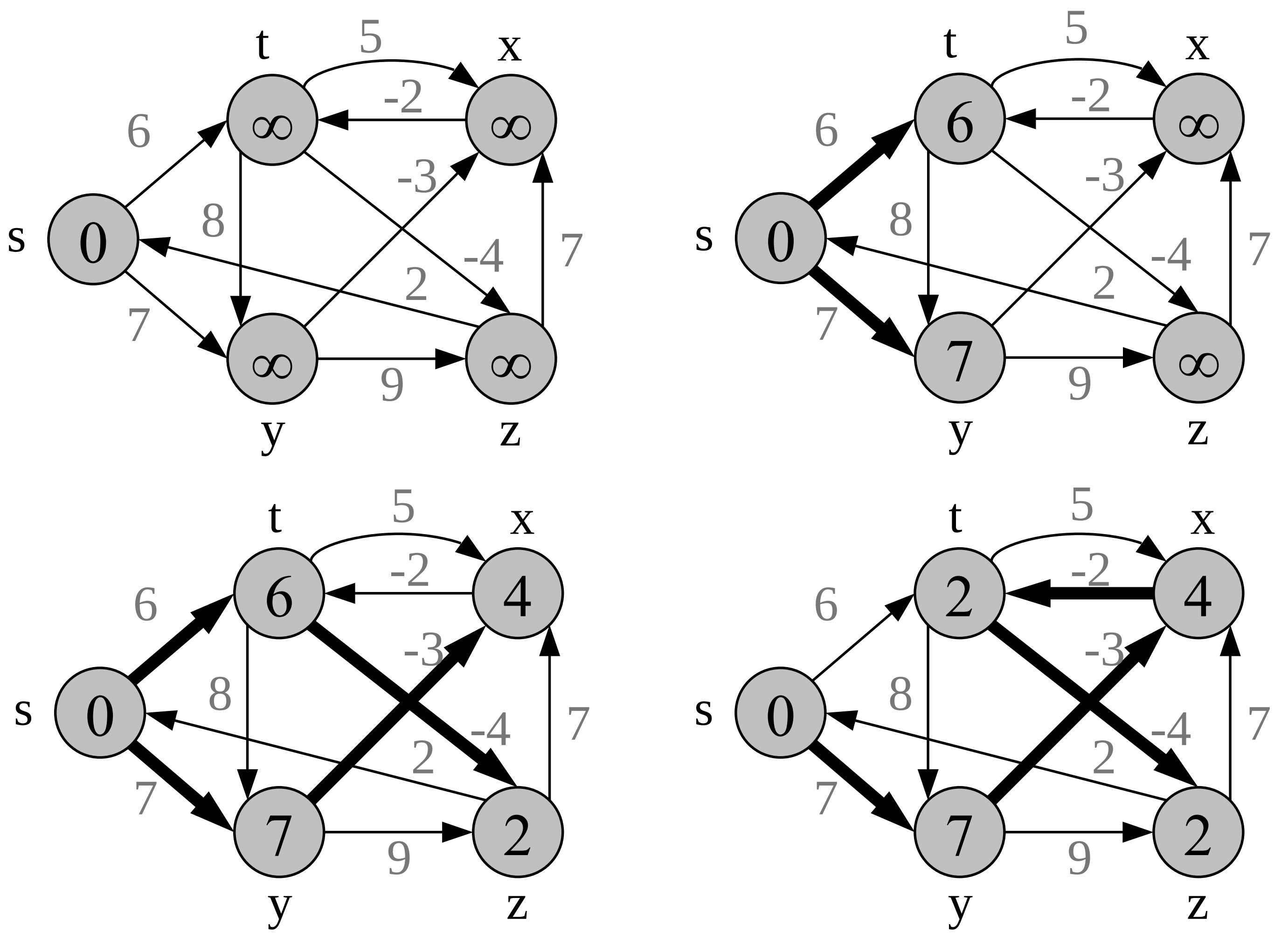
Bellman-Ford Algorithm

- 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

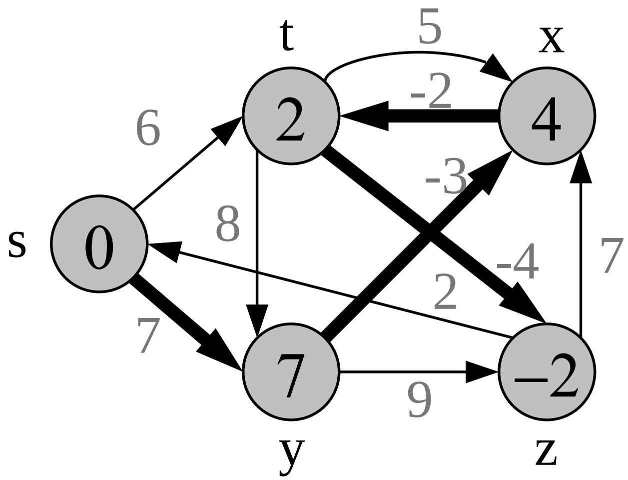
Bellman-Ford Algorithm

```
Bellman-Ford(G,w,s)
for each v ∈ V[G]
   d[v] ← ∞
d[s] ← 0
π[s] ← NIL
for i ← 1 to |V[G]|-1 do
    for each edge (u,v) ∈ E[G] do
        Relax (u,v,w)
for each edge (u,v) ∈ E[G] do
    if d[v] > d[u] + w(u,v) then return false
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

Shortest-Path in DAG's

- Finding shortest paths in DAG's is much easier, because it is easy to find an order in which to do relaxations – Topological sorting!

```
DAG-Shortest-Paths(G,w,s)
for each v ∈ V[G]
   d[v] ← ∞
d[s] ← 0
topologically sort V[G]
for each vertex u, taken in topological order do
   for each vertex v ∈ Adj[u] do
      Relax(u,v,w)
```

Shortest-Path in DAG's (2)

- Running time:

  $\Theta(V+E)$ – only one relaxation for each edge, V times faster than Bellman-Ford

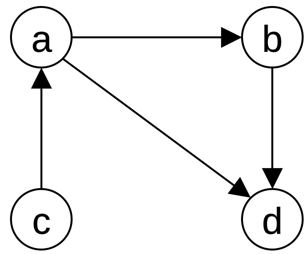
Next....

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

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

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

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.

$$\left(A^{m+1}\right)_{ij} = \sum_{k=1}^{q} (A^m)_{ik} A_{kj}$$

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.

$$\left(W^{m+1}\right)_{ij} = \min_{k=1}^{q}\left((W^m)_{ik} + W_{kj}\right)$$

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

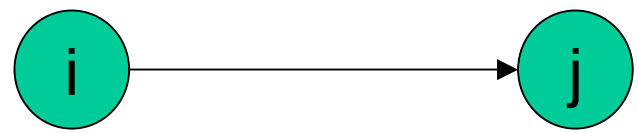
- Instead of increasing the length of the path allowed at each step, suppose that we increase the number of vertices that can be used in forming such paths.

- 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.

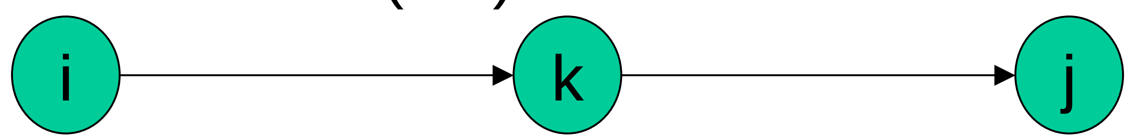
- Note that $D^{(0)} = W$. How can we calculate $D^{(n+1)}$ in terms of $D^{(n)}$?

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)

$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
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

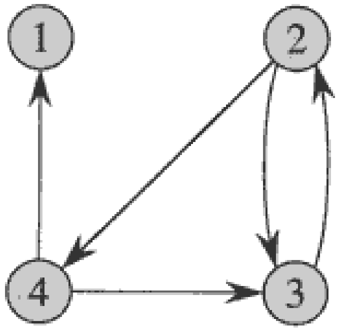
Transitive closure algorithm

Very similar to Floyd Warshall:

TRANSITIVE-CLOSURE(G)

$n \leftarrow |V[G]|$
for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do if $i = j$ or $(i, j) \in E[G]$
      then $t_{ij}^{(0)} \leftarrow 1$
      else $t_{ij}^{(0)} \leftarrow 0$
for $k \leftarrow 1$ **to** n
  do for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
      do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
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.