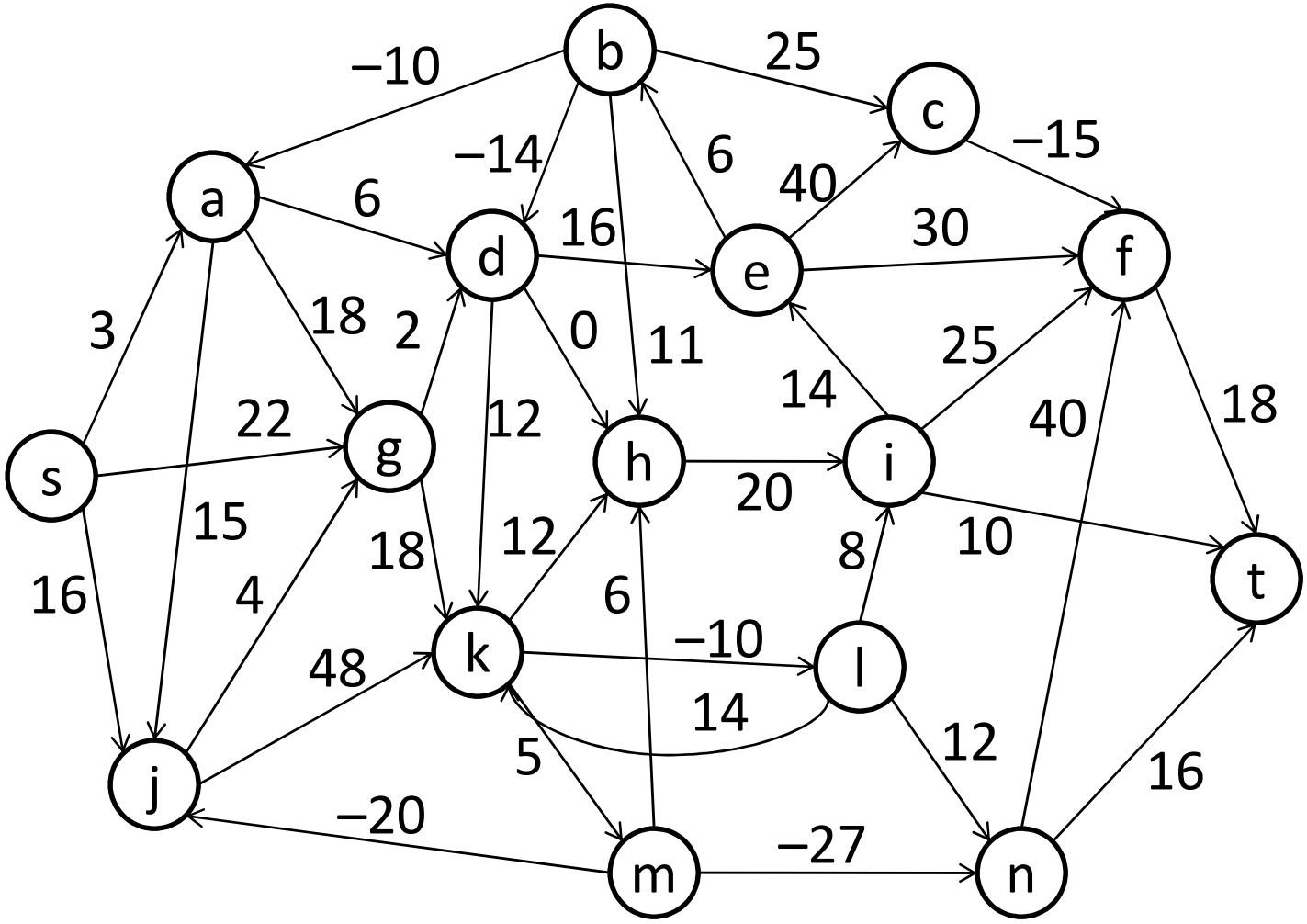
COS 423 Lecture 8
Shortest Paths

Directed graph with arc weights



path weight = sum of arc weights along path

Goal: find a minimum-weight path from *s* to *t*, for given pairs of vertices *s*, *t*

weights: costs, travel times, lengths,…

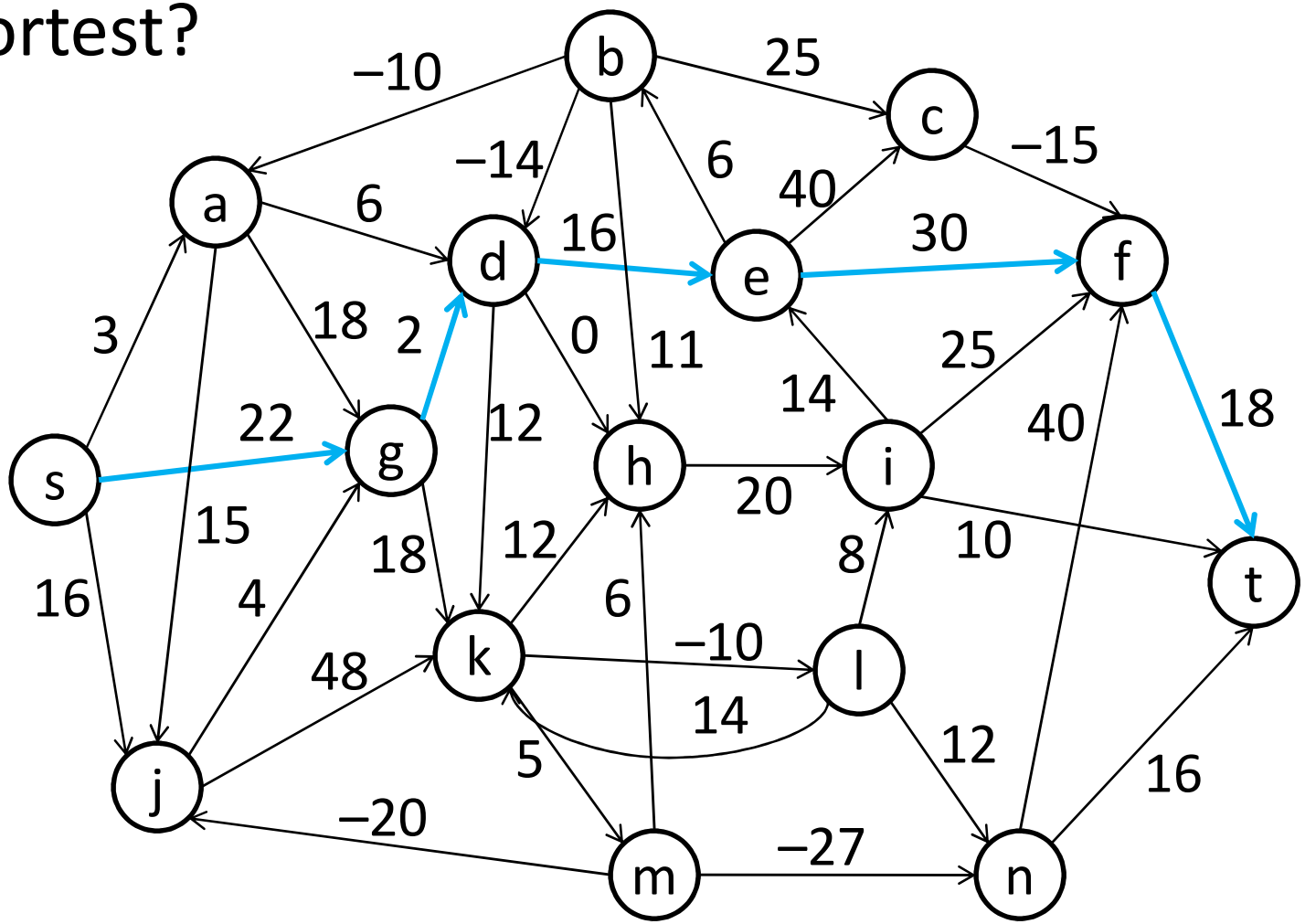
Henceforth think of weights as lengths;

a minimum-weight path is *shortest* (but we allow negative lengths)

Path s, g, d, e, f, t

Length 22 + 2 + 16 + 30 + 18 = 88

Shortest?



Versions of shortest path problem

Single pair: from one *s* to one *t*
Single source: from one *s* to each possible *t*
Single sink: from each possible *s* to one *t*
All pairs: from each possible *s* to each possible *t*

Single-source problem is central:

- equivalent to single-sink problem (reverse arc directions)
- all pairs = *n* single-source problems
- single-pair algorithms at least partially solve a single-source (or single-sink) problem

Special cases

Graph is undirected

Graph is planar

No negative arcs

No cycles

Negative cycles

A *negative cycle* is a cycle whose total length is negative.

If there are no negative cycles and there is *some* path from *s* to *t* (*t* is *reachable* from *s*), then there is a shortest path from *s* to *t* that is *simple* (it contains no repeated vertices): deletion of a cycle from the path does not increase the length of the path.

If a negative cycle is reachable from *s*, then there are arbitrarily short paths from s to every vertex on the cycle: just repeat the cycle.

If there are negative cycles, the problem of finding a shortest *simple* path is NP-hard.

Revised goal: Find a shortest path from *s* to *t* for each of the given pairs *s*, *t*, or find a negative cycle.

In some applications, negative cycles are good, and the goal is to find one.

Currency arbitrage: find a money-making cycle of currency trades

$1 = ¥83.1724 ¥1 = £0.00741115

£1 = €1.18694 €1 = $1.3668

Does trading $ for ¥ for £ for € for $ (or some other cycle of trades) make money?

Graph: vertices are currencies, arcs are currency conversions, weights are exchange rates

Value of cycle: product of exchange rates around cycle

money-making $\leftrightarrow$ value > 1

Transform: arc length = –lg(exchange rate)

value > 1 $\leftrightarrow$ cycle length < 0

Notation

$G = (V, A)$: graph with vertex set V and arc set A

$n = |V|$, $m = |A|$, assume $n > 1$

s: source vertex for single-source or single-pair problem

t: target vertex for single-sink or single-pair problem

(v, w): arc from v to w

$c(v, w)$ = length of arc (v, w)

$c(P)$ = length of path P

Single–source problem: find shortest paths from *s* to each vertex reachable from *s*, or find a negative cycle reachable from *s*.

Method: *iterative improvement*. For each vertex *v*, maintain the length $d(v)$ of the shortest path from *s* to *v* found so far. Look for shorter paths by repeatedly examining arcs (v, w). If $d(v) + c(v, w) < d(w)$, there is a shorter path to *w*: decrease $d(w)$ to $d(v) + c(v, w)$. Stop when no such improvement is possible.

Labeling algorithm

for $w \in V$ **do** $d(w) \leftarrow \infty$; $d(s) \leftarrow 0$;

while $\exists (v, w) \in A \ni d(v) + c(v, w) < d(w)$ **do**

$label(w)$: $d(w) \leftarrow d(v) + c(v, w)$

Each iteration of the **while** loop is a *labeling step*.

(The Operations Research literature calls such a step a *relaxation* of (v, w).)

Lemma 1: The labeling algorithm maintains the invariant that if $d(w) < \infty$, w is reachable from s, and $d(w)$ is the length of a path from s to w.

Proof: For each assignment to $d(w)$ we define a path $P(w, d(w))$ from s to w of length $d(w)$, as follows: $P(s, 0)$ is the path consisting of vertex s and no arcs; if $d(w) \leftarrow d(v) + c(v, w)$, the path $P(w, d(w))$ is $P(v, d(v))$ followed by (v, w).

Theorem 1: If the algorithm stops, $d(w)$ if finite is the length of a shortest path from s to w; $d(w) = \infty \leftrightarrow w$ is unreachable from s.

Proof: Let P be a shortest path from s to w. If the algorithm stops, $d(x) + c(x, y) \geq d(y)$ for every arc (x, y) on P. Summing over all arcs on P gives $c(P) \geq d(w) - d(s) \geq d(w)$, since $d(s) \leq 0$. By Lemma 1, $d(w) = c(P)$ (and $d(s) = 0$). In particular, if w is reachable from s, $d(w) < \infty$.

Theorem 1 implies that if there is a negative cycle reachable from *s*, the labeling algorithm never stops.

If there are no negative cycles, a stronger version of Lemma 1 holds:

Lemma 2: If there are no negative cycles, each path $P(w, d(w))$ is simple.

Proof: Suppose the lemma is false. Let $P(w, d(w))$ be the first such path defined that is not simple, and let $d(w) \leftarrow d(v) + c(v, w)$ be the corresponding assignment. Then $P(v, d(v))$ is simple but contains w. Thus $P(v, d(v))$ is $P(w, d')$ followed by P', where $P(w, d')$ is a path corresponding to an earlier assignment and P' is a path from w to v. Then $d' > d(v) + c(v, w)$ and $c(P') = d(v) - d'$. The cycle P' followed by (v, w) has length $d(v) - d' + c(v, w) < 0$, a contradiction.

Theorem 2: If there are no negative cycles, the algorithm stops.

Proof: By Lemma 2, the number of labeling steps is at most the number of simple paths from *s*.

The bound on the number of steps given by the proof of Theorem 2 is exponential, and indeed the labeling algorithm takes exponential time in the worst case. To make the algorithm efficient, we must choose the order of steps carefully.

Before addressing how to choose labeling steps, we extend the algorithm to find shortest paths, not just their lengths.

To do this, we maintain a *parent* $p(w)$ for each vertex w: $p(w)$ is the next-to-last vertex on the shortest path to w found so far.

Labeling algorithm with parents

for $w \in V$ **do** $\{d(w) \leftarrow \infty;\ p(w) \leftarrow \text{null}\};\ d(s) \leftarrow 0;$

while $\exists (v, w) \in E \ni d(v) + c(v, w) < d(w)$ **do**

$\{d(w) \leftarrow d(v) + c(v, w);\ p(w) \leftarrow v\}$

Labeling algorithm with parents

for $w \in V$ **do** $\{d(w) \leftarrow \infty;\ p(w) \leftarrow \text{null}\};\ d(s) \leftarrow 0;$

while $\exists (v, w) \in E \ni d(v) + c(v, w) < d(w)$ **do**

$\{d(w) \leftarrow d(v) + c(v, w);\ p(w) \leftarrow v\}$

Lemma 3: If $p(w) \neq \text{null}$, $d(p(w)) + c(p(w), w) \leq d(w)$.

Proof: Just after a step that decreases $d(w)$, $d(p(w)) + c(p(w), w) = d(w)$. Until $d(w)$ decreases again, $p(w)$ does not change, and $d(p(w))$ cannot increase.

Lemma 4: If the algorithm stops and $p(w) \neq \text{null}$, $d(p(w)) + c(p(w), w) = d(w)$.

Proof: If p(w) ≠ null and(p(w) + c(p(w), w) ≠ d(w), d(p(w)) + c(p(w), w) < p(w) by lemma 3, so the algorithm does not stop.

Lemma 5: Any cycle of arcs $(p(x), x)$ is negative.

Proof: Suppose a labeling step creates a cycle C of such arcs by assigning $p(w) \leftarrow v$. Consider the state just before the step. For any vertex $x \neq w$ on the cycle, $c(p(x), x) \leq d(x) - d(p(x))$ by Lemma 3. Also, $c(v, w) < d(w) - d(v)$. Summing these inequalities over all arcs on C gives $c(C) < 0$. (All terms on the right side cancel.)

Theorem 3: If there are no negative cycles, the arcs ($p(v)$, v) form a tree T rooted s (no arc enters s, one arc enters each vertex other than s, and there are no cycles) containing exactly the vertices reached from s. When the algorithm stops, T is a *shortest path tree* (*SPT*): every path in T is shortest.

Proof: Immediate from Theorems 1 and 2 and Lemmas 4 and 5.

Corollary 1: G contains either a shortest path tree rooted at s or a negative cycle reachable from s.

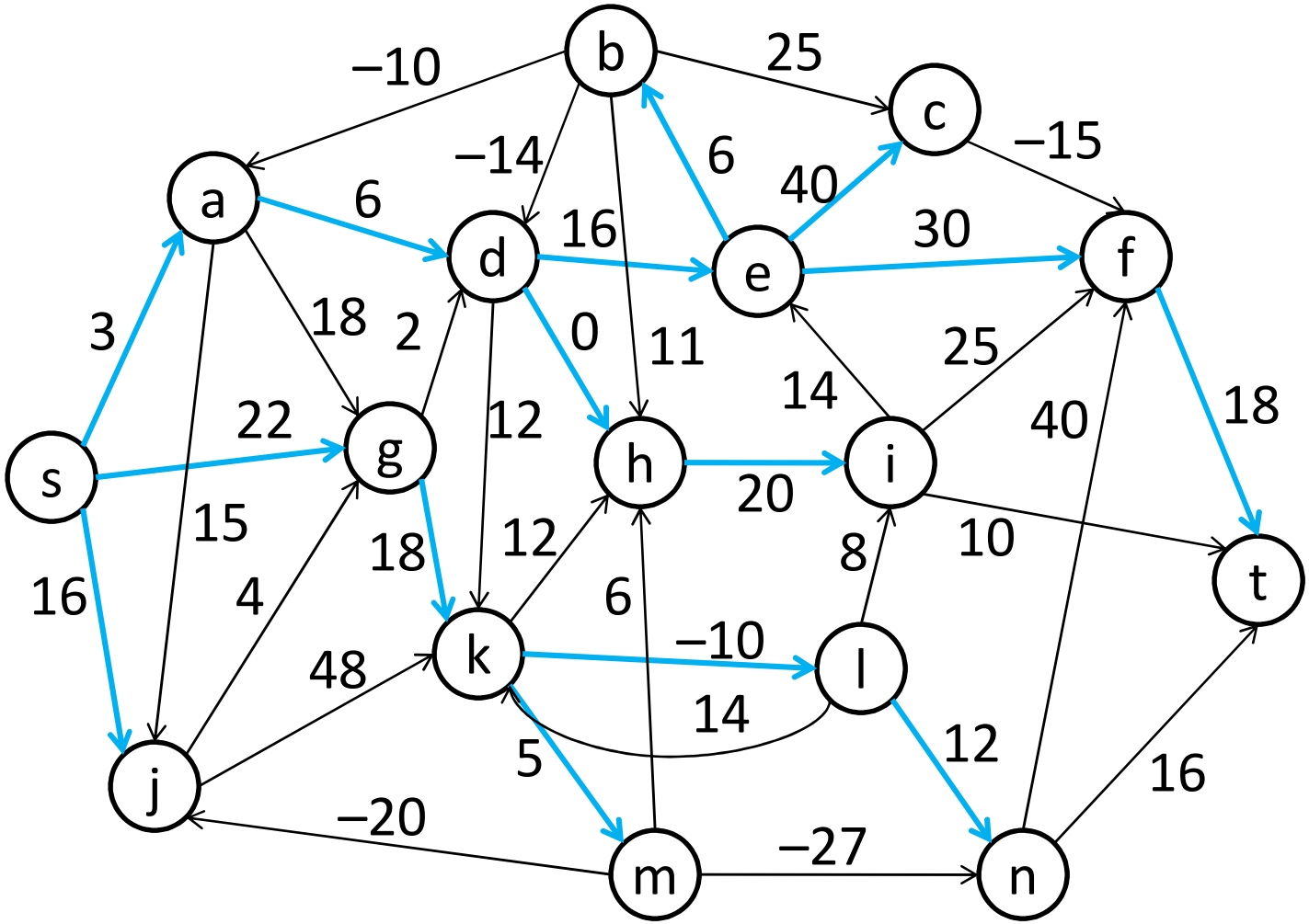
Interlude: Shortest Path Tree Verification

Suppose we are given a graph G and a tree T rooted at s whose arcs are in G. How can we test whether T is a shortest path tree(*SPT*) of G?

The labeling algorithm provides an O(m)-time test: for each vertex w in G, compute $d(w)$ as follows: $d(s) = 0$, $d(w) = d(p(w)) + c(p(w), w)$ where $p(w)$ is the parent of w in T, $d(w) = \infty$ if w is not in T. Then T is an SPT if and only if for every arc (v, w) in G, $d(v) + c(v, w) \geq d(w)$.

SPT Verification

Given a spanning tree, is it an SPT?



To make the shortest path algorithm efficient, we begin by doing the labeling steps vertex-by-vertex instead of arc-by-arc.

We partition the vertices into three sets: *U* (unlabeled), *L* (labeled), and *S* (scanned). Each vertex in *U* has not been reached from s; each vertex in *L* may have outgoing arcs that give shorter paths, and each vertex in *S* has been reached and its arcs have been checked since its distance last changed.

The scanning algorithm

for $w \in V$ **do** $\{d(w) \leftarrow \infty; p(w) \leftarrow \text{null}\}$; $d(s) \leftarrow 0$;

$U \leftarrow V - \{s\}$; $L \leftarrow \{s\}$; $S \leftarrow \{\ \}$;

while some $v \in L$ **do** *scan*(v):

{**for** each arc (v, w) out of v **do**

if $d(v) + c(v, w) < d(w)$ **then**

$\{d(w) \leftarrow d(v) + c(v, w)$; $p(w) \leftarrow v$;

move w to $L\}$;

move v to $S\}$

$v \in U \leftrightarrow d(v) = \infty$

If $d(v) + c(v, w) < d(w)$, then v is in L.

Proof: By induction on #steps. True initially.
Once $d(v) + c(v, w) \geq d(w)$, can only become false if $d(v)$ decreases, in which case v is moved to L.

→ Labeling and scanning algorithm is correct

Graph Representation

For each vertex, store the set of outgoing arcs

Store arc sets in lists, or in arrays, which can be subarrays of one big array

Array representation saves space (no pointers), improves locality of access

Efficient scanning orders

General graph:

Breadth-first scanning (Bellman-Ford)

L = *queue*, add new labeled vertices to back

Non-negative arc lengths:

Shortest-first scanning (Dijkstra)

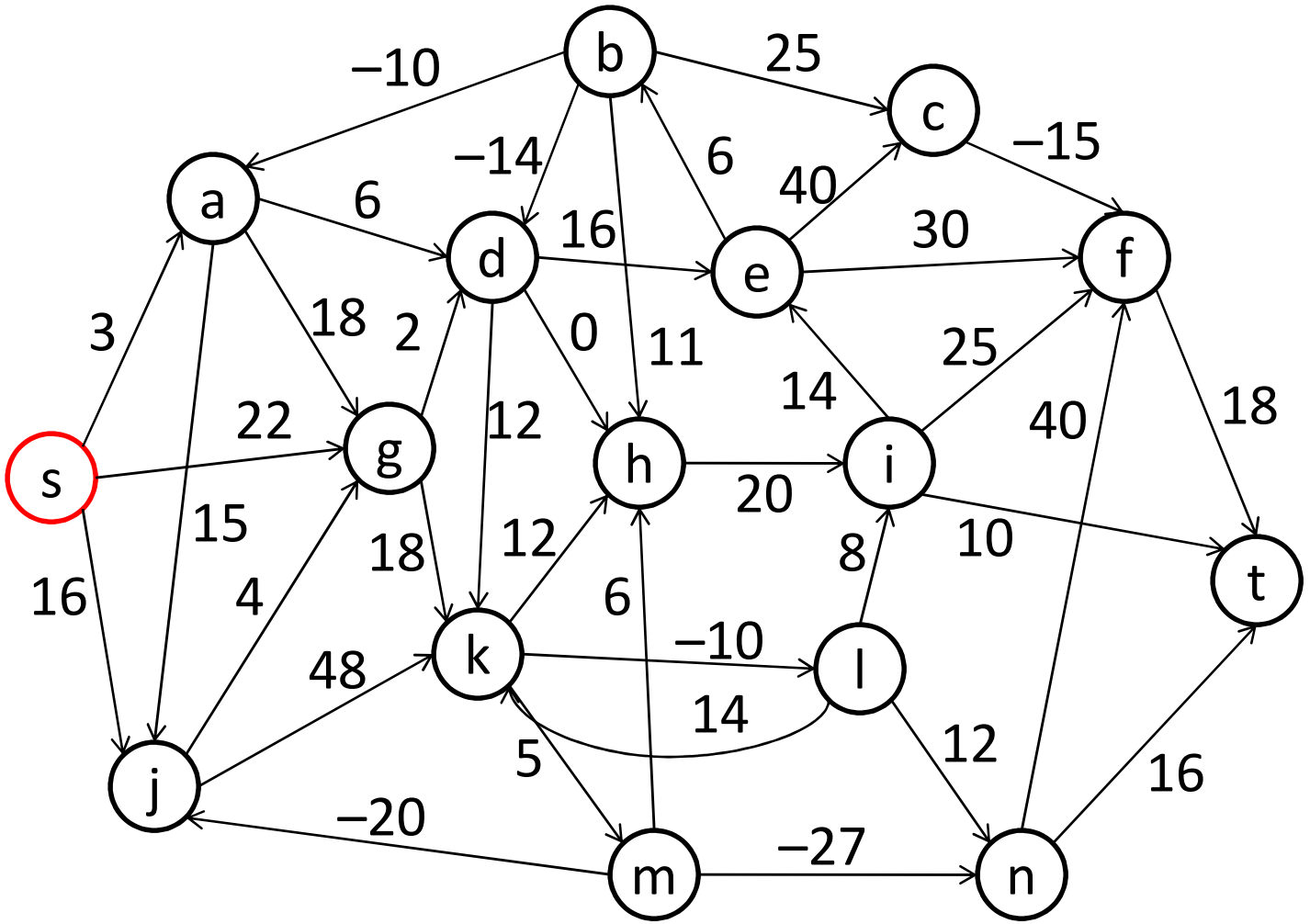
L = *heap*, distances are keys

Acyclic graph:

Topological scanning

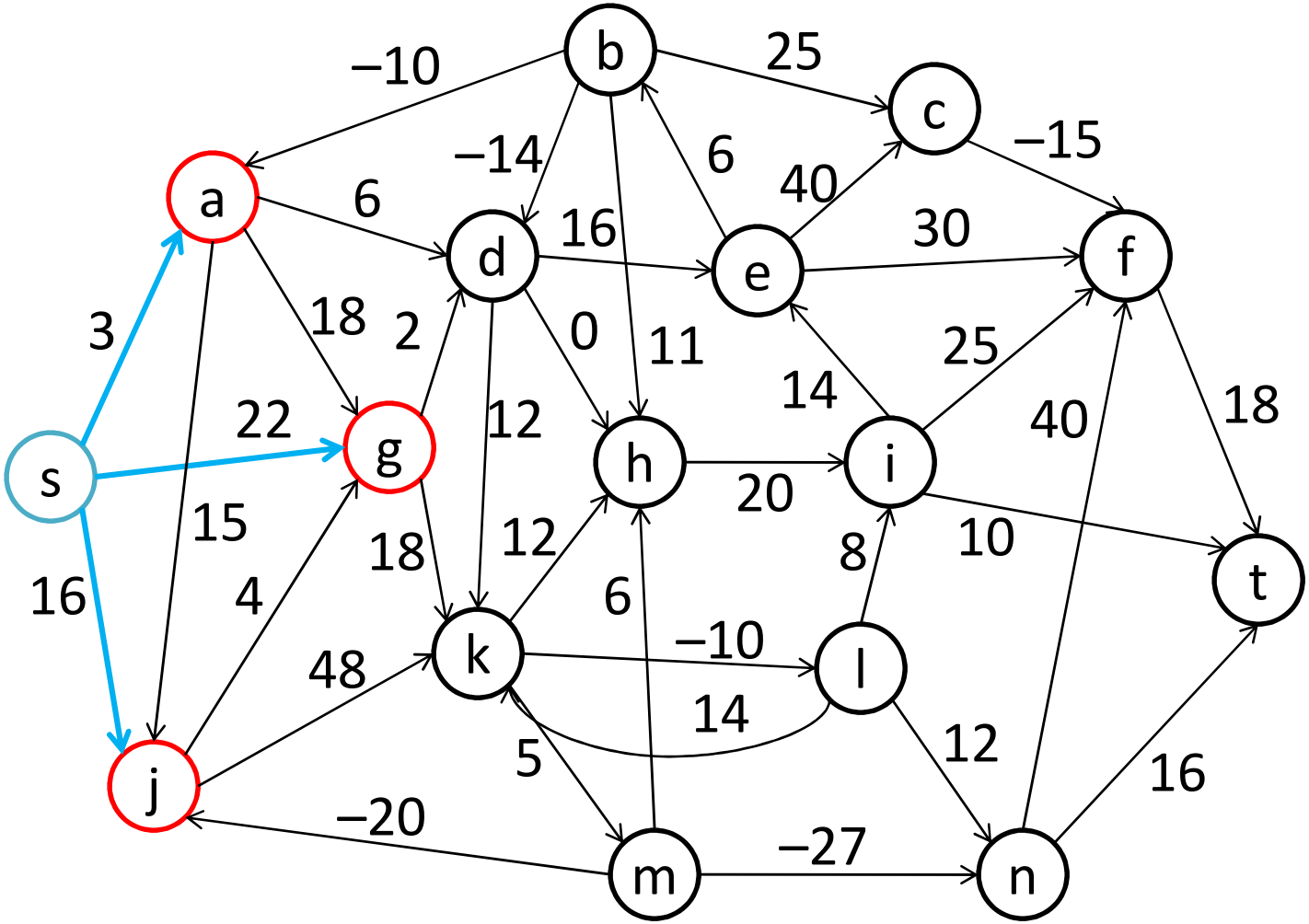
Breadth-first scanning

L = s:0 *scan* s



S = s:0 *scan* a

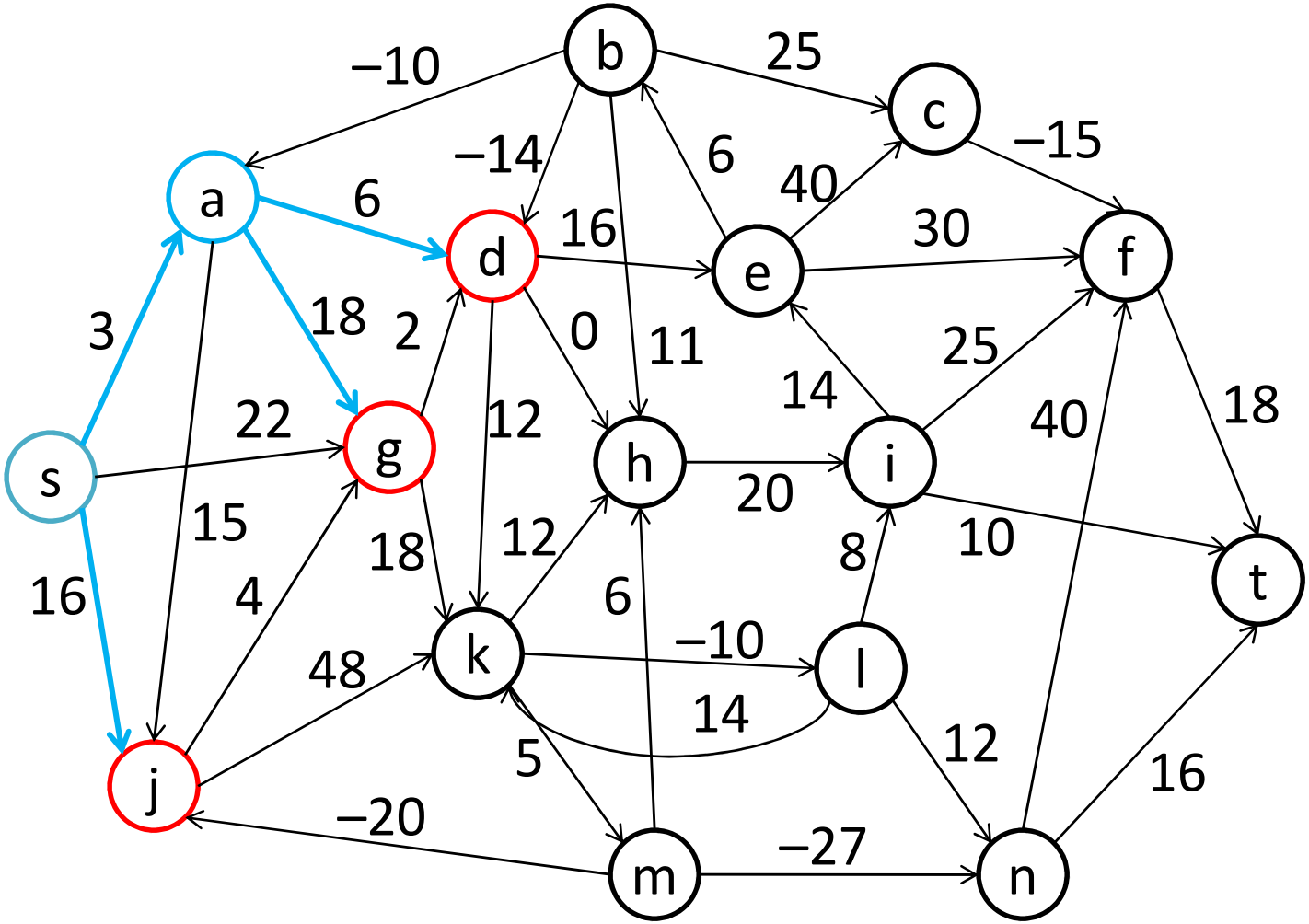
L = a:3, g:22, j:16



S = s:0, a:3

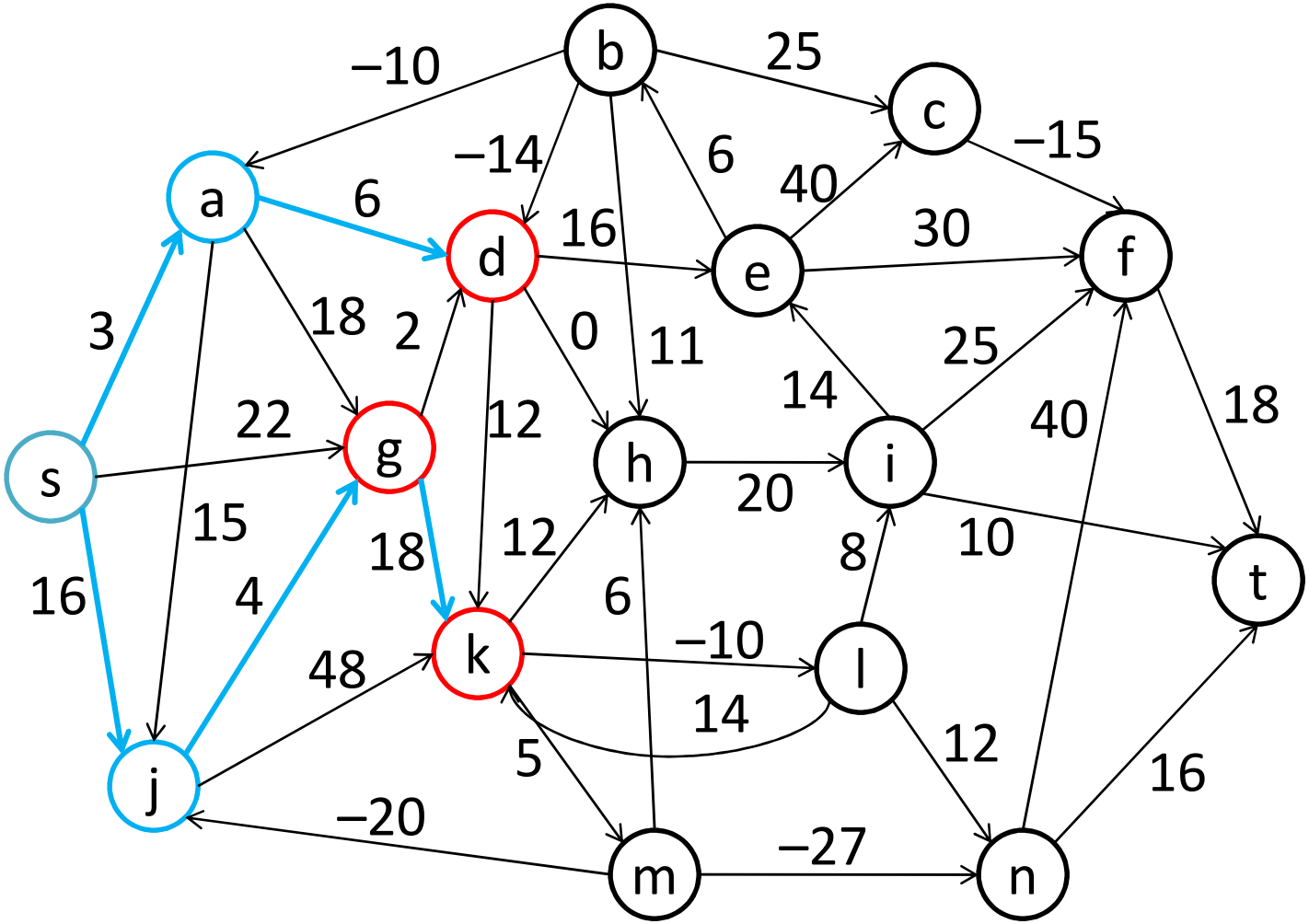
L = g:21, j:16, d:9

scan g, *scan* j



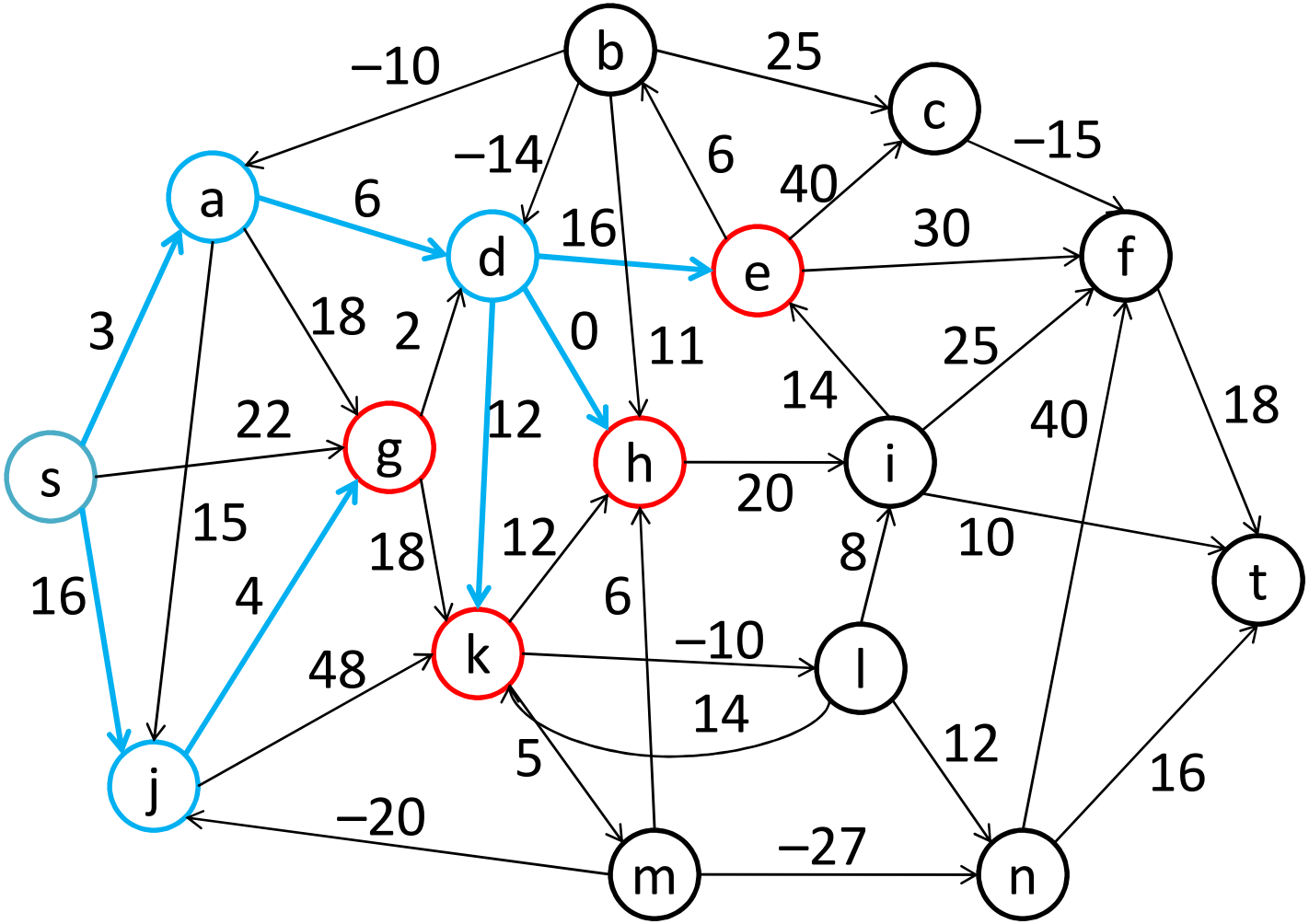
S = s:0, a:3, j:16 *scan* d

L = d:9, k:39, g:20



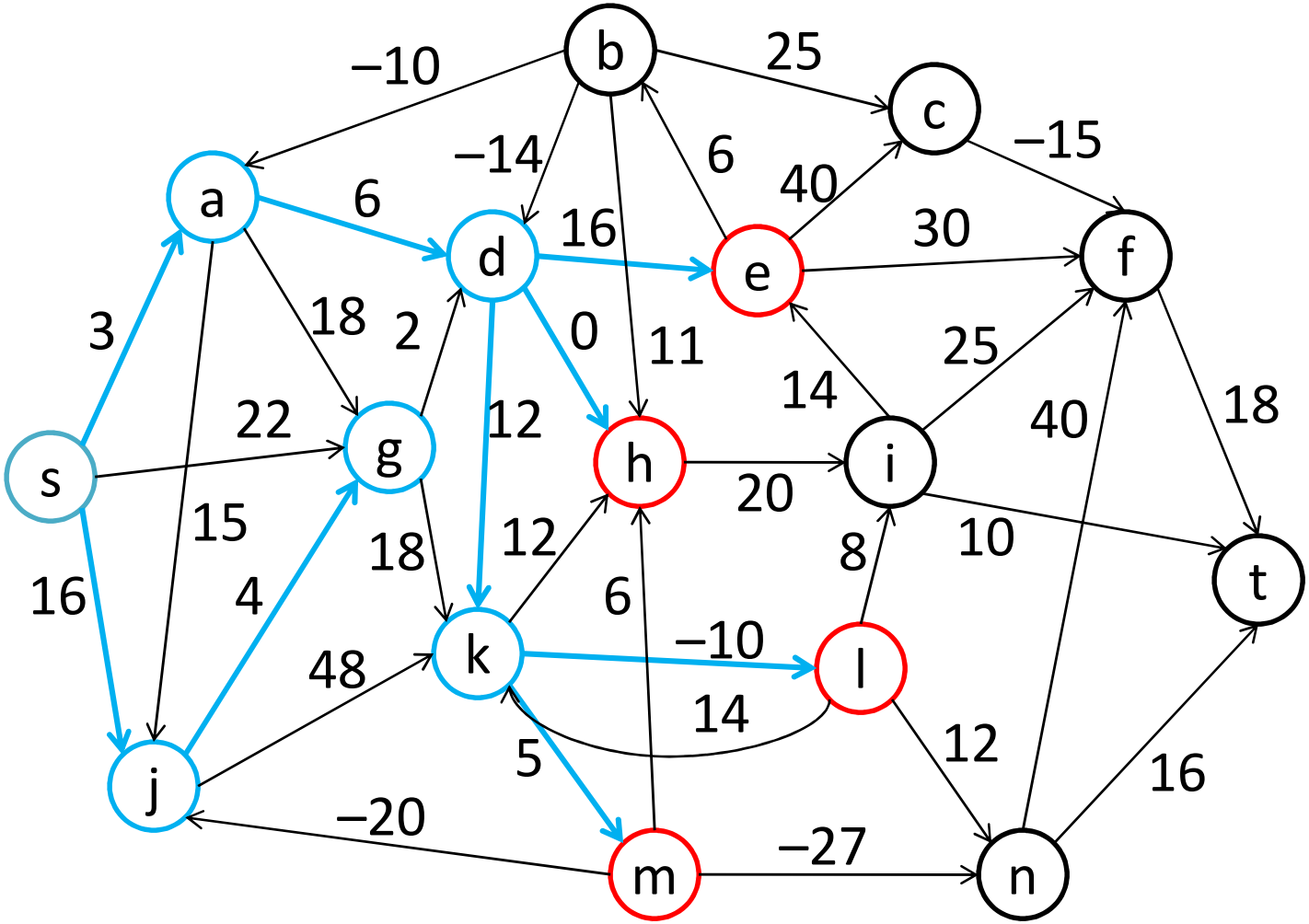
S = s:0, a:3, j:16, d:9 *scan* k, *scan* g

L = k:21, g:20, h:9, e:25



S = s:0, a:3, j:16, d:9, k:21, g:20

L = h:9, e:25, i:11, m:26 …



Running time of breadth-first scanning

Define *passes* through the queue:

pass 0 = scanning of s

pass $k + 1$ = scanning of all vertices added to the queue during pass k

After pass k, each vertex having a shortest path from s of k arcs has correct distance

→ all distances correct after pass $n - 1$

→ algorithm stops after $\leq n$ passes, or never

Each pass scans each vertex at most once

→ $O(nm)$ time

Generalized breadth-first scanning

The O(*nm*) time bound holds as long as all vertices added to *L* in pass *k* are scanned before any vertex added to *L* in pass *k* + 1.

Two-set implementation: Add newly scanned vertices to *L'*. Once *L* is empty, move all vertices in *L'* to *L*. Scan vertices in *L* in arbitrary order.

Negative cycle detection

Lazy: count passes. If count exceeds *n*, stop: there must be a negative cycle. Such a cycle can be found by following parent pointers. (Exercise: prove this.)

Eager: test for a cycle of parent pointers during each labeling step.

How?

If $d(v) + c(v, w) < d(w)$, follow parent pointers from v until reaching w (negative cycle found) or s

This method takes $\Theta(n)$ time per labeling step, increasing the running time to $\Theta(n^2 m)$

Better: The predecessor pointers define a tree rooted at s. Instead of starting at v and visiting its ancestors looking for w, start at w and visit its descendants looking for v

Why better?

$$d(v) + c(v, w) < d(w)$$

We disassemble the subtree rooted at *w* as we traverse it looking for *v*. Each deletion of a vertex from *T* is preceded by a labeling of the same vertex; this labeling pays for the deletion. Furthermore each vertex other than *w* deleted from *T* will be labeled again later: its distance is not minimum. This method is *subtree disassembly*.

We store the vertices of T in a doubly-linked circular list in *preorder* (all descendants of a vertex are consecutive, each child follows its parent, not necessarily consecutively), using pointers a (*after*) and b (*before*).

If $d(v) + c(v, w) < d(w)$, we visit the descendants of w, deleting each from T. If v is visited, a cycle exists. If not, we decrease $d(w)$ and reinsert w into T as a child of v. Optionally, for each vertex $x \neq w$ deleted from T, we can decrease $d(x)$ by almost as much as the decrease in $d(w)$.

Scanning with subtree disassembly and distance updates

for $w \in V$ **do** $\{d(w) \leftarrow \infty$; $p(w) \leftarrow$ null; $b(w) \leftarrow$ null$\}$;
$d(s) \leftarrow 0$; $U \leftarrow V - \{s\}$; $L \leftarrow \{s\}$; $S \leftarrow \{\}$; $a(s) \leftarrow s$; $b(s) \leftarrow s$;
while some $v \in L$ **do**
  {**for** each arc (v, w) out of v **do**
    $\{\delta \leftarrow d(w) - d(v) - c(v, w)$;
     if $\delta > 0$ **then**
       $\{d(w) \leftarrow d(w) - \delta$; $p(w) \leftarrow v$; move w to L;
        $x \leftarrow b(w)$; $b(w) \leftarrow$ null; $y \leftarrow a(w)$;
        while $b(p(y)) =$ null **do**
          if $y = v$ **then stop**: negative cycle
          else $\{d(y) \leftarrow d(y) - \delta + \varepsilon$; $b(y) \leftarrow$ null; $y \leftarrow a(y)\}$;
       $a(x) \leftarrow y$; $b(y) \leftarrow x$; $a(w) \leftarrow a(v)$; $b(a(w)) \leftarrow w$;
       $b(w) \leftarrow v$; $a(v) \leftarrow w\}\}$; move v to $S\}$

Scanning with subtree disassembly and distance updates

for $w \in V$ **do** $\{d(w) \leftarrow \infty$; $p(w) \leftarrow$ null; $b(w) \leftarrow$ null$\}$;
$d(s) \leftarrow 0$; $U \leftarrow V - \{s\}$; $L \leftarrow \{s\}$; $S \leftarrow \{\ \}$; $a(s) \leftarrow s$; $b(s) \leftarrow s$;
while some $v \in L$ **do**
  {**for** each arc (v, w) out of v **do**
    $\{\delta \leftarrow d(w) - d(v) - c(v, w)$;
     if $\delta > 0$ **then**
       $\{d(w) \leftarrow d(w) - \delta$; $p(w) \leftarrow v$; move w to L;
        $x \leftarrow b(w)$; $b(w) \leftarrow$ null; $y \leftarrow a(w)$;
        while $b(p(y)) =$ null **do**
          if $y = v$ **then stop**: negative cycle
          else $\{d(y) \leftarrow d(y) - \delta + \varepsilon$; $b(y) \leftarrow$ null; $y \leftarrow a(y)\}$;
       $a(x) \leftarrow y$; $b(y) \leftarrow x$; $a(w) \leftarrow a(v)$; $b(a(w)) \leftarrow w$;
       $b(w) \leftarrow v$; $a(v) \leftarrow w\}\}$; move v to $S\}$

The *before* pointers indicate whether a vertex is in T: x in $T \leftrightarrow b(x) \neq$ null

If the algorithm detects a negative cycle, the cycle can be found by following parent pointers from v (or from w).

If $w = s$, there is a negative cycle.

After the **while** loop, y is the vertex after the last (now deleted) descendant of w in preorder.

The **while** loop removes each proper descendant y of w from T and decreases its distance by $\delta - \varepsilon$, where $\delta = d(w) - d(v) + c(v, w)$ and ε is the smallest representable positive number (if arc lengths are integers, $\varepsilon = 1$); we cannot reduce $d(y)$ by an additional ε, because we must make sure that y is labeled again before the algorithm stops. (Why?)

Not only does subtree disassembly detect negative cycles eagerly, it speeds up the scanning algorithm in practice: it results in fewer scans of vertices whose distance from *s* is not yet minimum. Thus it is an important heuristic even if eager negative cycle detection is not needed.

Theorem 4: Breadth-first scanning with subtree disassembly, with or without distance updates, runs in O(*nm*) time and stops as soon as a cycle of parent pointers exists.