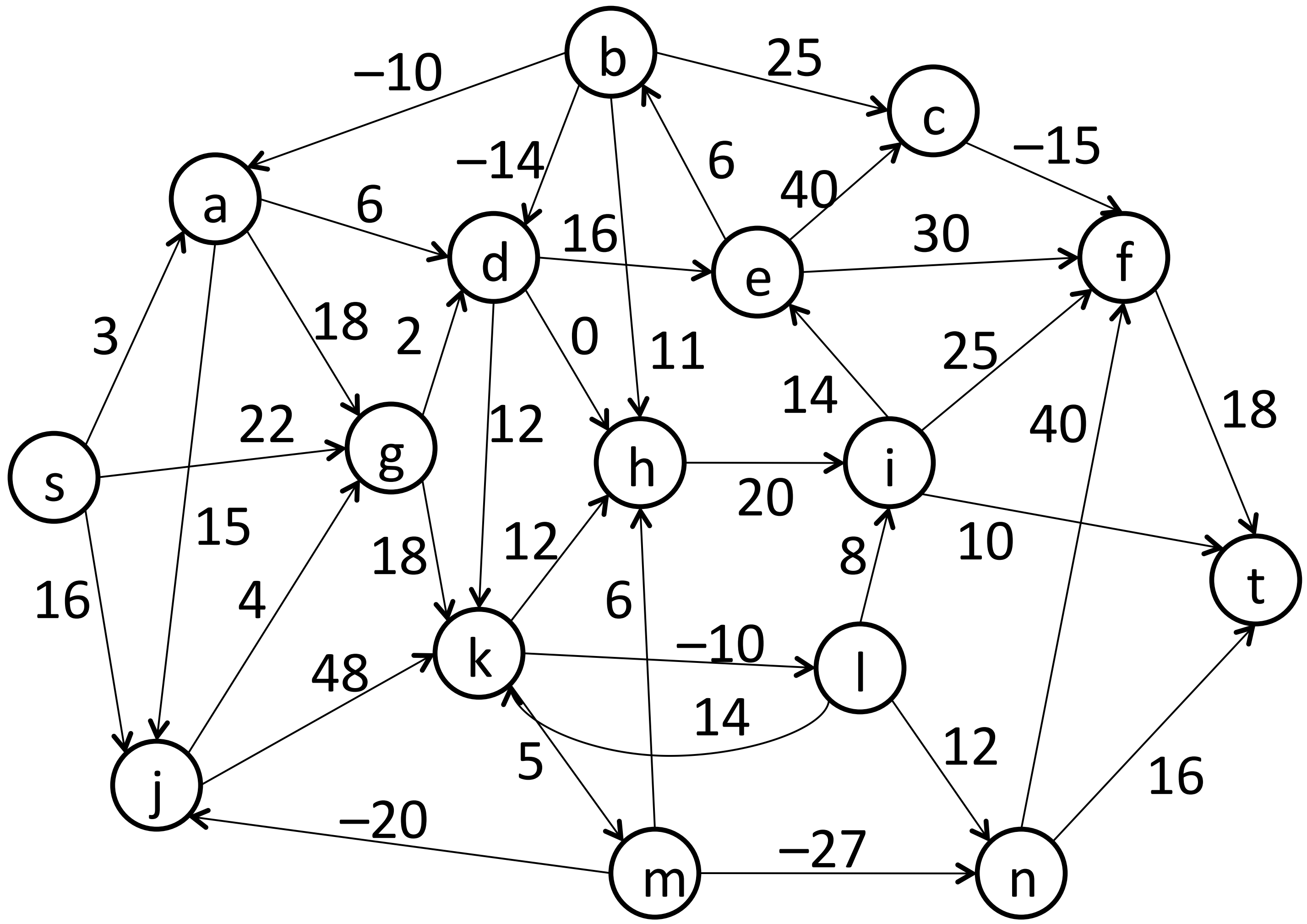
COS 423 2/10/14
Shortest Paths

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*

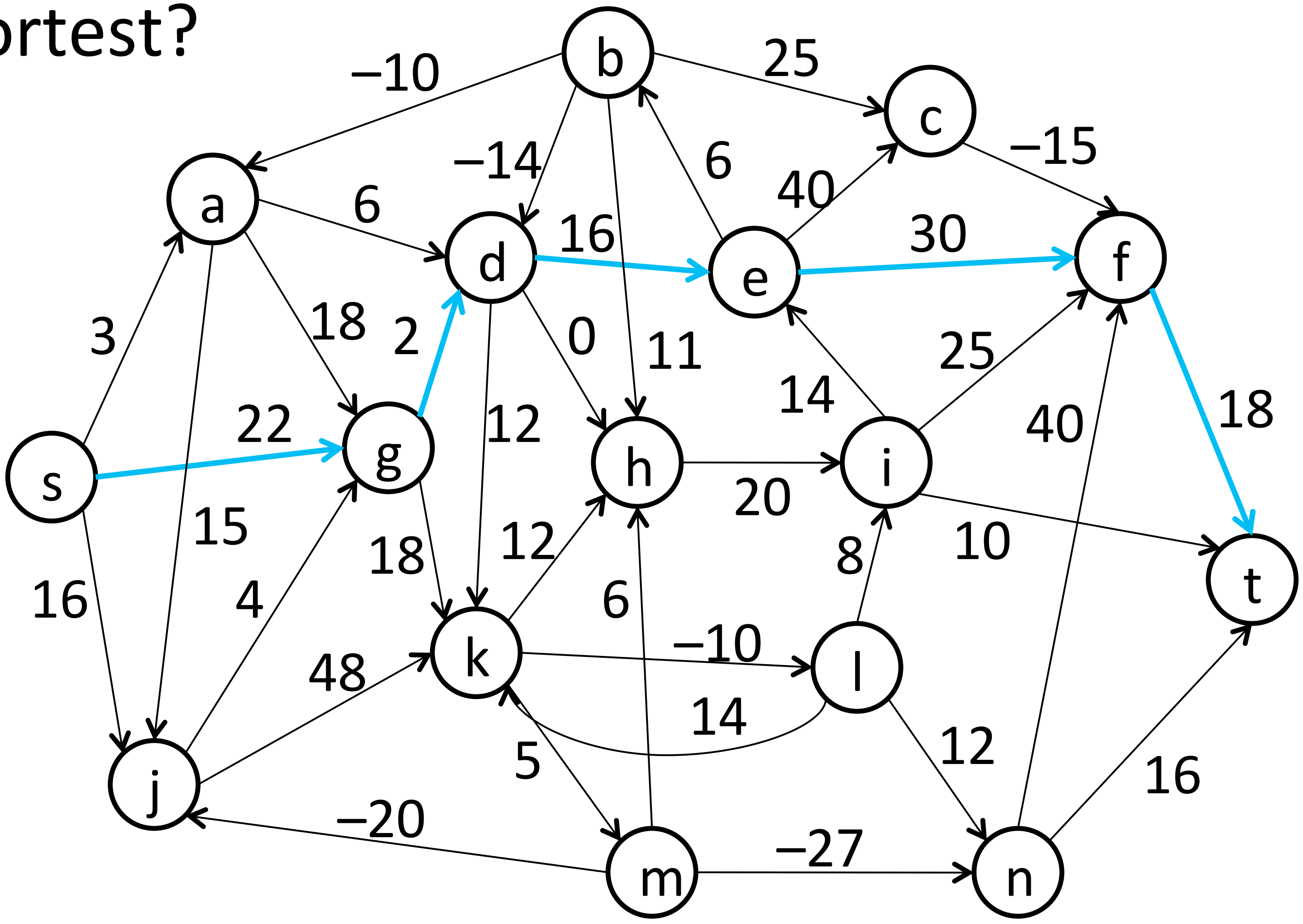
weights: costs, travel times, lengths,…

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

No negative arcs

No cycles

Planar graph

Road network

Public transportation network

Public and private transportation network

(car, train, bus, airplane, walking…)

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 $n > 1$, $m > n - 1$

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 a *label* $d(v)$ equal to the length of a shortest path from *s* to *v* found so far. Look for shorter paths by repeatedly *scanning* arcs (v, w). If $d(v) + c(v, w) < d(w)$, a shorter path to *w* exists: reduce $d(w)$ to $d(v) + c(v, w)$. Stop when no such improvement is possible.

The Labeling Algorithm

```
**for** $w \in V$ **do** $d(w) \leftarrow \infty$; $d(s) \leftarrow 0$;
**repeat** *scan* some arc $(v, w)$
**until** no scan can reduce a distance
**where** *scan*$(v, w)$:
    **if** $d(v) + c(v, w) < d(v, w)$ **then**
        $d(w) \leftarrow d(v) + c(v, w)$
```

(The Operations Research literature calls scanning (v, w) *relaxing* (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, and $d(w) = \infty \leftrightarrow w$ is unreachable from s.

Proof: Suppose the algorithm stops. Let P be any path from s to w. Then $d(x) + c(x, y) \geq d(y)$ 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 $d(s) \leq 0$. Thus if there is some path from s to w, $d(w) < \infty$. If d(w) < ∞, there is a path P^* from s to w of length $d(w)$ by Lemma 1. Since P is *any* path, P^* is a shortest path. If there is no path from s to w, then $d(w) = \infty$ by Lemma 1.

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*)) defined in the proof of Lemma 1 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 formed by 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. (Example?) To make the algorithm efficient, we must choose the order of scanning steps carefully.

The Labeling Algorithm with Parents

We can extend the algorithm to find shortest paths, not just their lengths, by maintaining 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.

```
initialize $p(s) \leftarrow$ null;
$scan(v, w)$: if $d(v) + c(v, w) < d(w)$ then
        $\{d(w) \leftarrow d(v) + c(v, w); p(w) \leftarrow v\}$
```

Lemma 3: If $p(w) \neq$ 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 $p(w) \neq$ null, $d(p(w)) + c(p(w), w) = d(w)$.

Proof: If $\mathrm{p(w)} \neq$ null and$(\mathrm{p(w)} + \mathrm{c(p(w), w)} \neq \mathrm{d(w)}$, $\mathrm{d(p(w))} + \mathrm{c(p(w), w)} < \mathrm{p(w)}$ 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 at 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.

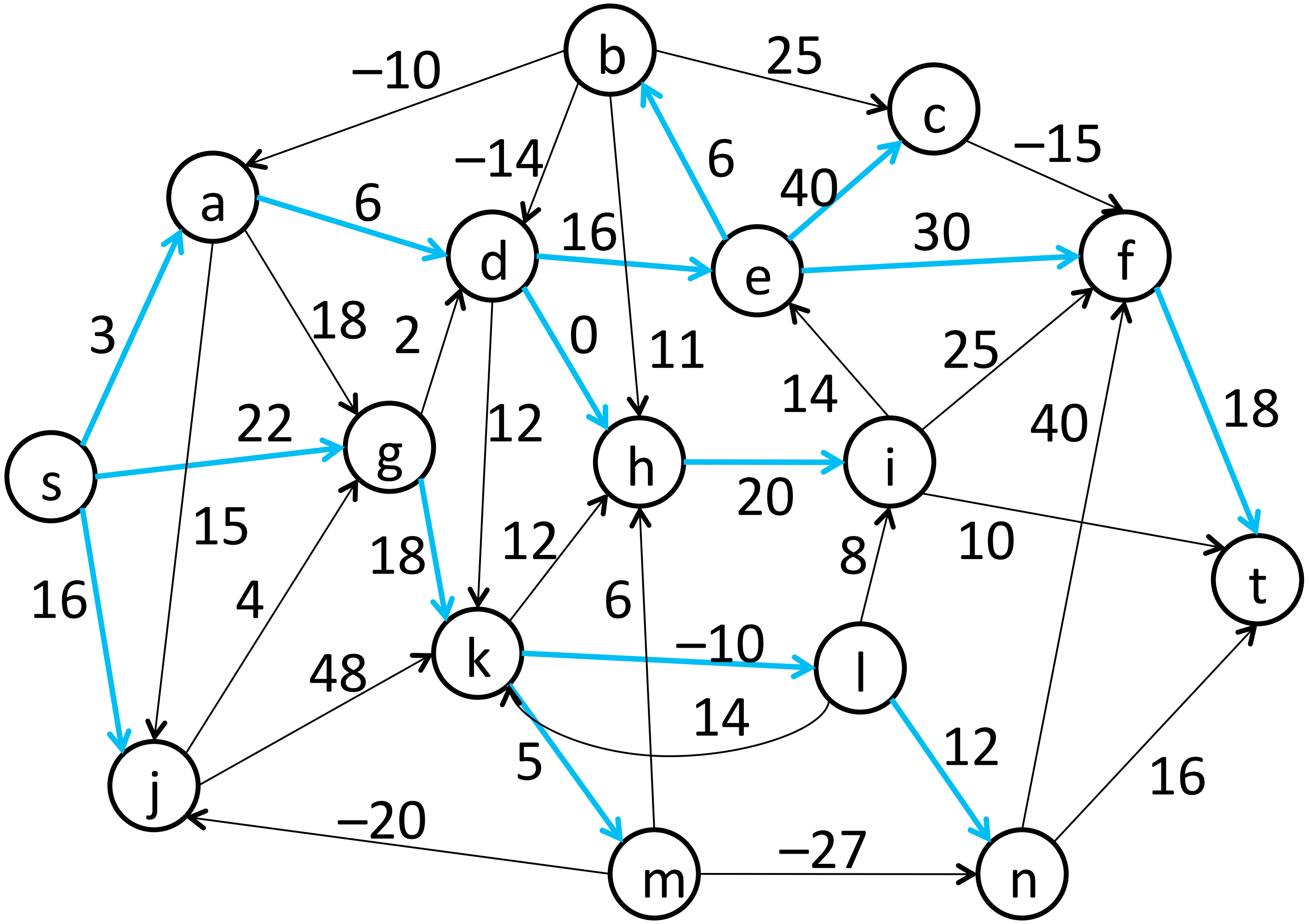
Interlude: Shortest Path Tree Verification

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?



Instead of starting with the labeling algorithm, we could have started with the verification algorithm and obtained the labeling algorithm from it: A failure of the verification test gives a shorter path to some vertex; the labeling algorithm consists of applying the verification test, updating a distance if it fails, and repeating until it succeeds.

Good Scanning Orders

General graphs:

Breadth-first (Bellman-Ford): scan an arc (v, w) with fewest previous scans

Non-negative arc lengths:

Shortest-first (Dijkstra): scan an unscanned (v, w) with d(v) minimum

Acyclic graphs:

Topological: scan an unscanned (v, w) with all arcs into v scanned

Breadth-First Labeling (Bellman-Ford)

```
for $w \in V$ do $d(w) \leftarrow \infty$; $d(s) \leftarrow 0$; $p(s) \leftarrow$ null;
repeat scan every arc
until no distance changes
```

Each pass can scan the arcs in any order. The algorithm stops when an entire pass changes no distances.

Running Time of Breadth-First Labeling

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

Each vertex has a shortest path of at most *n* – 1 arcs → all distances correct after pass *n* – 1

→ algorithm stops after ≤*n* passes, or never

Each pass scans each arc at most once

→ O(*nm*) time

Heuristics?

Breadth-first labeling, or the even simpler algorithm of scanning all the arcs $n - 1$ times, is often taught. (See COS 226.) But these simple versions of Bellman-Ford can do many arc scans that do not decrease a distance, or do decrease a distance but not to the minimum. Such arc scans are *useless*.

We want to reduce the number of useless arc scans, but without increasing the O(nm) worst-case time bound. We look at two heuristics for doing this.

The first does the scans vertex-by-vertex instead of arc-by-arc. It maintains a set of *labeled vertices* L: if $d(v) + c(v, w) < d(w)$ (arc (v, w) needs to be scanned), $v \in L$.

The Scanning Algorithm

```
**for** $w \in V$ **do** $d(w) \leftarrow \infty$; $d(s) \leftarrow 0$; $L \leftarrow \{s\}$;
**while** some $v \in L$ **do**
    {delete $v$ from $L$;
     $scan(v)$: **for** each arc $(v, w)$ **do**
        $scan(v, w)$:
            **if** $d(v) + c(v, w) < d(w)$ **then**
                {$d(w) \leftarrow d(v) + c(v, w)$; $p(w) \leftarrow v$;
                 **if** $w$ not in $L$ **then** add $w$ to $L$}};
```

Lemma 6: 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 added to L.

→ The 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

Either representation stores the graph in O(*n* + *m*) = O(*m*) space.

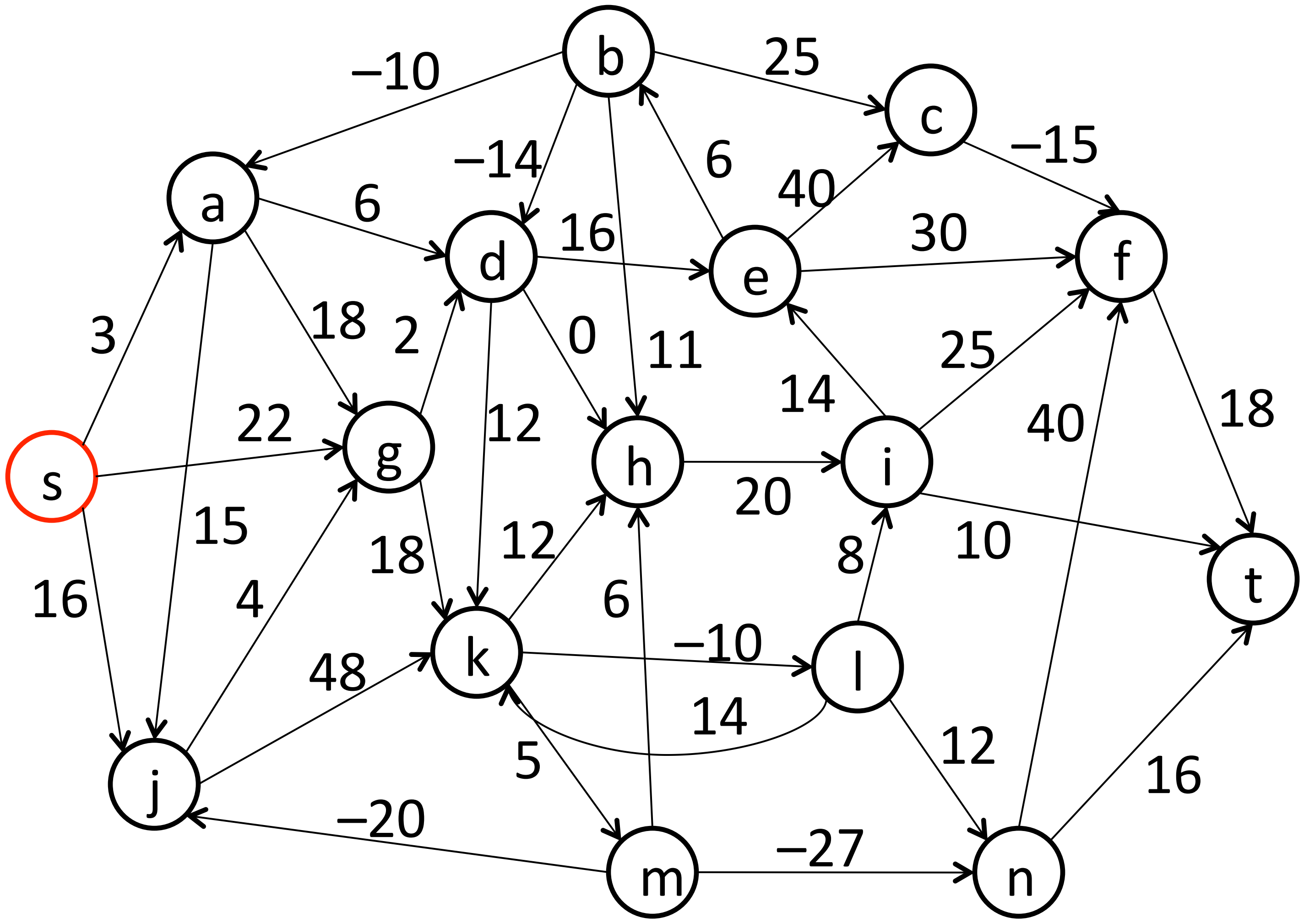
The Breadth-First Scanning Algorithm

Maintain *L* as a queue (first-in-first-out): add to the back, delete from the front.

Generalized breadth-first scanning via two sets: Add newly scanned vertices to *L′*. Once *L* is empty, move all vertices in *L′* to *L*. Scan vertices in *L* in arbitrary order.

Breadth-first scanning

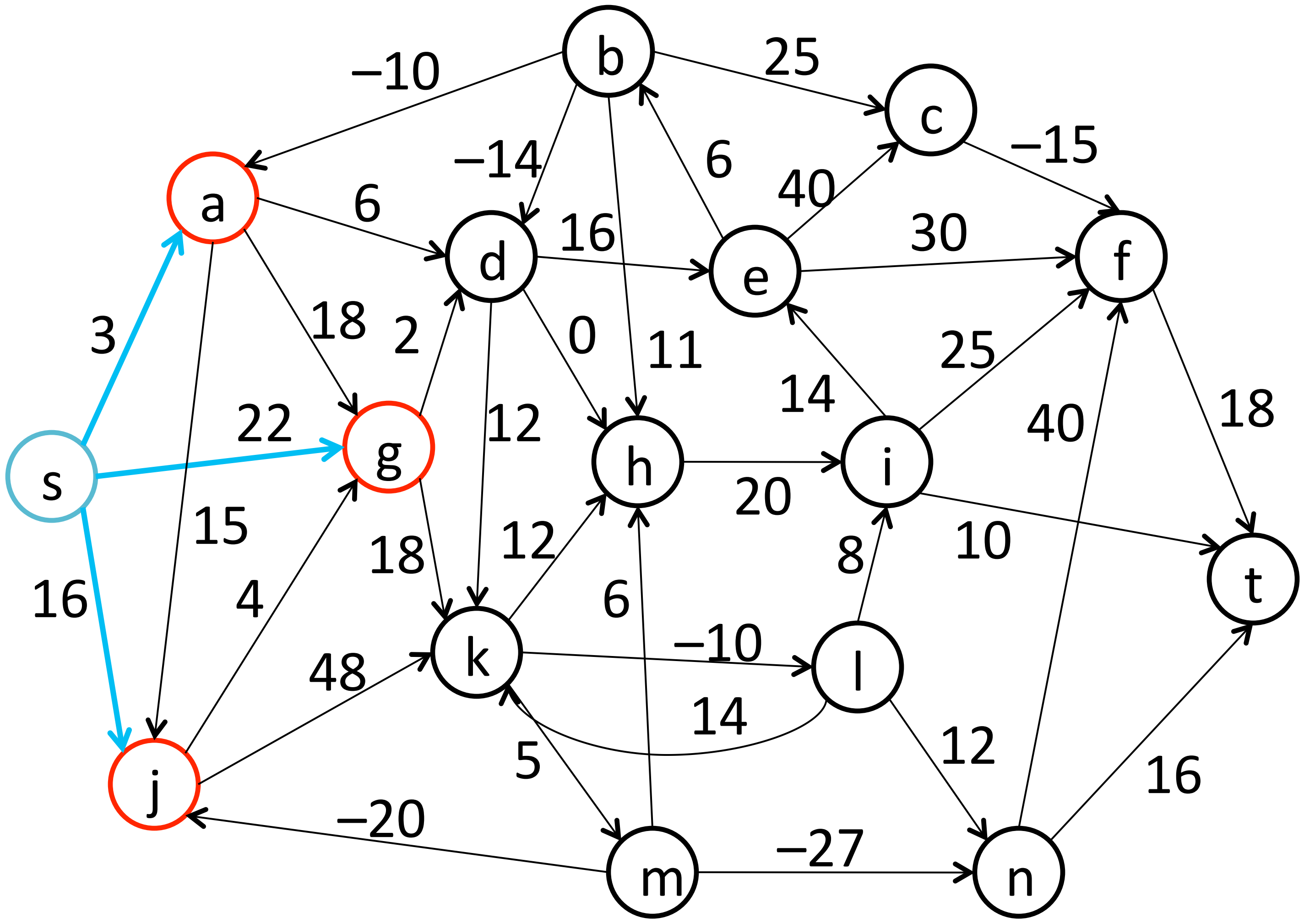
L = s:0 *scan* s



s:0

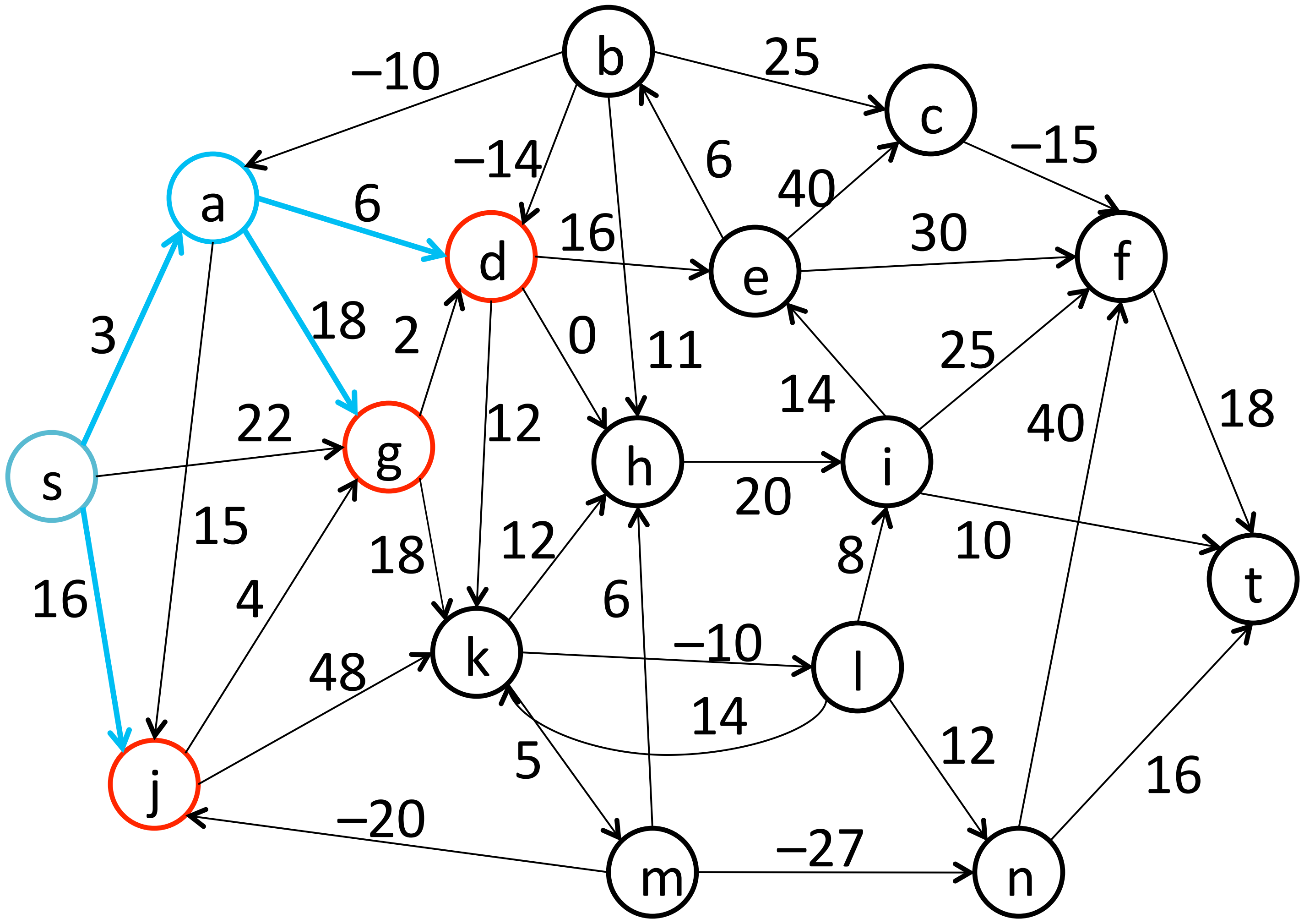
scan a

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



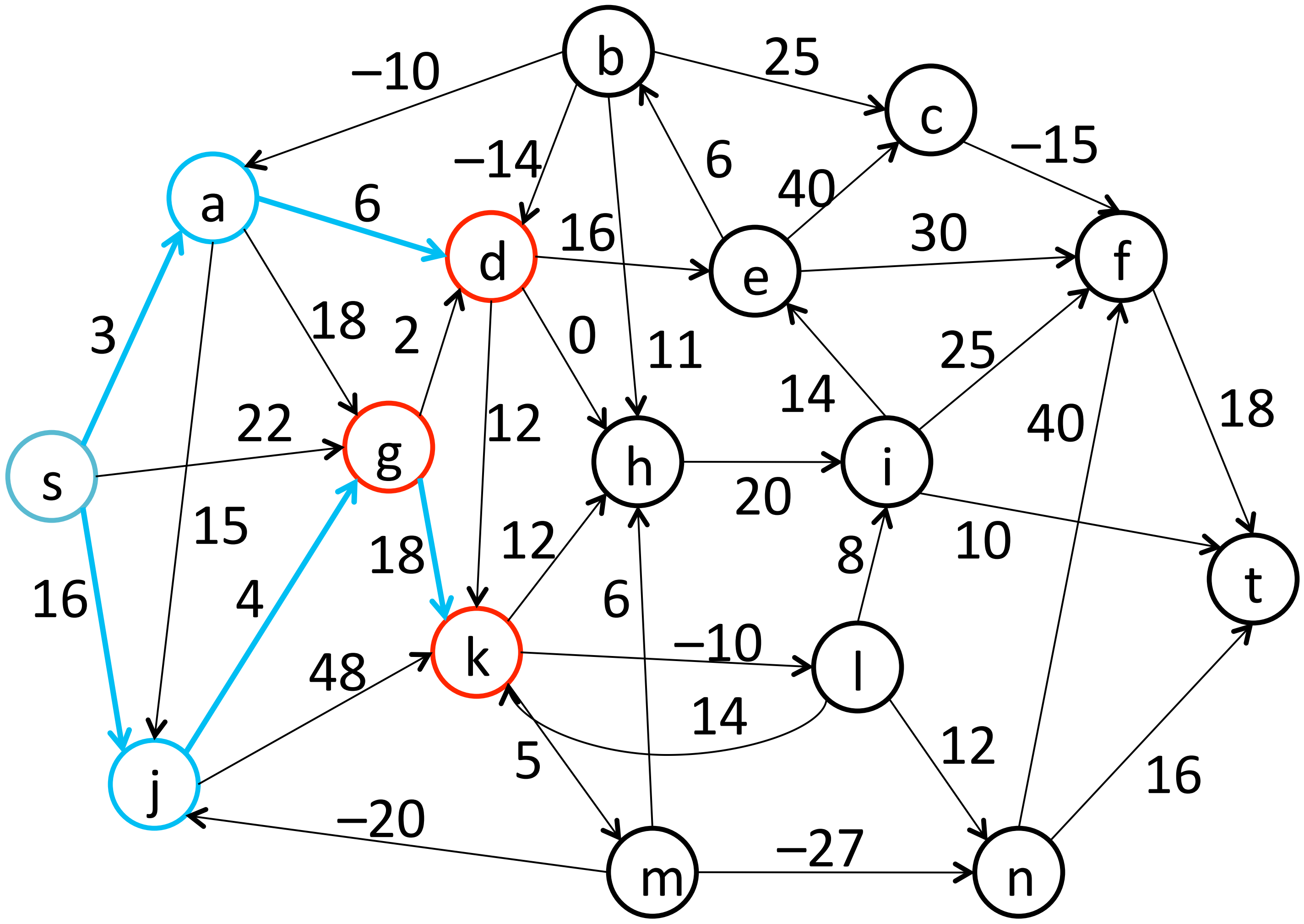
s:0, a:3 *scan* g, *scan* j

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



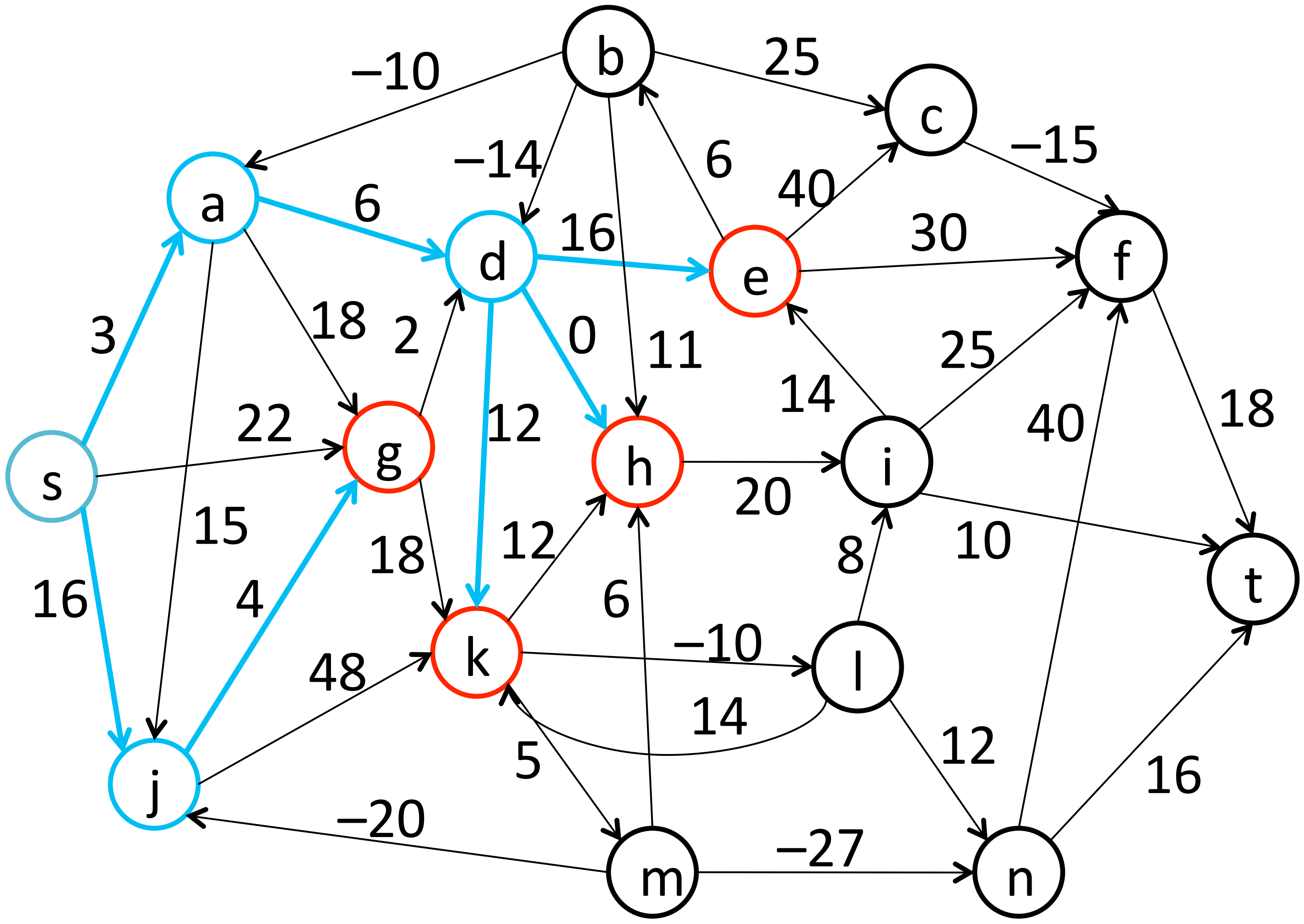
s:0, a:3, j:16 *scan* d

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



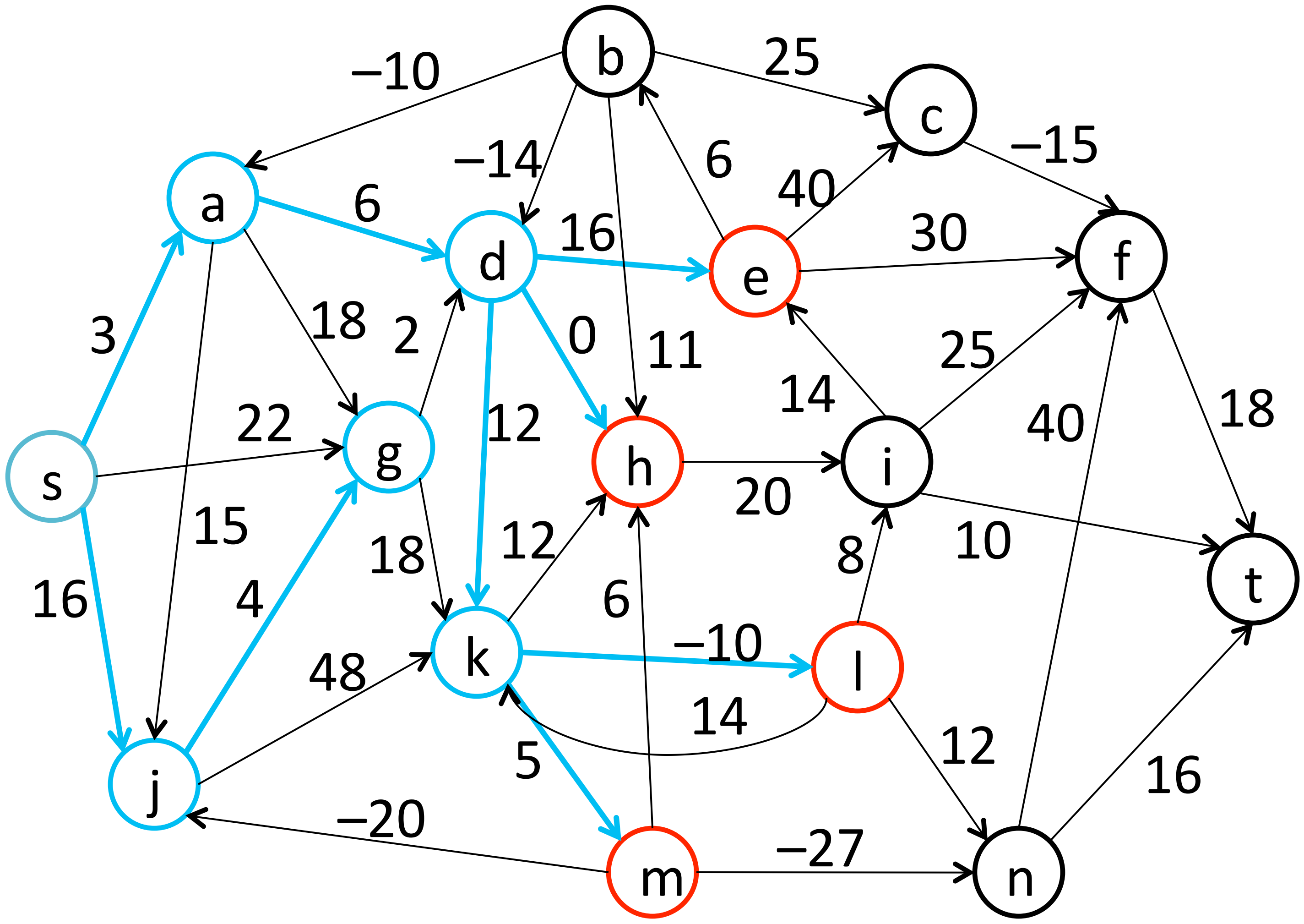
s:0, a:3, j:16, d:9   *scan* k, *scan* g

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



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

L = h:9, e:25, i:11, m:26 …



Running Time of (Generalized) Breadth-First Scanning

Same analysis as breadth-first labeling:

Define passes through L:

- pass 1 = scanning of s
- pass $k + 1$ = scanning of vertices added to L in pass k

After pass k, all distances to vertices with shortest paths of k arcs or fewer are correct

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

Each arc is scanned at most once per pass

→ O(*nm*) time

Breadth-first scanning can do many fewer arc scans than breadth-first labeling even though both have the same worst-case time bound and both do the same number of passes: the latter scans *every* arc during every pass, the former scans only those from vertices in *L*.

What we have learned

Either there is a shortest path tree (SPT) rooted at *s* or there is a negative cycle reachable from *s*.

Breadth-first labeling or breadth-first scanning will find an SPT rooted at *s* or a negative cycle reachable from *s* in at most *n* passes and O(*nm*) time.

Breadth-first labeling is simpler but can do many more useless arc scans: use breath-first scanning.