CSI 403: Design and Analysis of Algorithms

Shortest Paths – Chapter 24

J Marques de carvalho

Today

- Project-3
- Project-4
- Graphs
  - Single-source shortest-paths
    - Dijkstra
    - Bellman-Ford

Shortest Paths - Introduction

- Thus far, we have seen how to search through a graph, using DFS & BFS.
- These were unweighted graphs
  - This implies that there is no particular difference traversing along one edge versus another
- Now we look at how to determine the shortest path from one vertex to another, taking edge weights into account as well

The Shortest Path Problem

- Given a weighted, directed graph $G = (V, E)$, and a weight function $w: E \rightarrow \boldsymbol{R}$, the weight $w(p)$ of a path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is the sum of the weights of the path's edges:

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

- The shortest-path weight from the ***source*** vertex u to the ***destination*** vertex v is

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v & \text{if a path from } u \text{ to } v \text{ does exist} \\ \infty & \text{if no path from } u \text{ to } v \text{ exists} \end{cases}$$

The Shortest Path Problem

- Weights can be interpreted as distance, or some other “cost” – fuel (usage), time, etc.
- Weights can also be negative (fuel fill-up, battery charge, etc.)
  - We’ll handle negative weights a little later
- The Breadth-First Search (Ch. 22) is a shortest-path algorithm that works on unweighted graphs (graphs in which all edges are assumed to have a weight of 1)

Variants of the Shortest Path Problem

- The ***single-source shortest-path problem***: Given a graph $G = (V, E)$, find the shortest path from a source vertex $s \in V$ to each vertex $v \in V$

- ***Single-destination shortest-paths problem***: Find a shortest path to each destination vertex t from each vertex v. This is just the single-source shortest path problem above with the directions of the edges reversed.

Variants of the Shortest Path Problem

- ***Single-Pair Shortest Path problem***: Find a shortest path from a specific source vertex u to a specific destination vertex v. Solving the single-source problem with u as the source also solves this problem.
- ***All-pairs shortest-paths problem***: Find a shortest path from u to v for every pair of vertices u and v. We could solve the single-source problem from all vertices, but there are better ways (Chapter 25).

Optimal Substructure

- Shortest-path algorithms count on the property that the shortest path from u to v also contains the shortest paths among the vertices between u and v.

A Bit More Formally, …

- Lemma 24.1: Given a weighted, directed graph $G = (V, E)$ with weight function $w: E \rightarrow \mathbf{R}$
- Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k, and for any i and j such that $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j.
- Then p_{ij} is a shortest path from v_i to v_j.

The proof (1)

- Proof: If we break up the path p from v_i to v_k into pieces:

$$v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$$

  Then we have $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$.

- If there exists some *other* path p'_{ij} from vertex v_i to vertex v_j with weight $w(p'_{ij}) < w(p_{ij})$, then we have a new path from v_1 to v_k:

$$v_1 \xrightarrow{p_{1i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$$

The Proof (2)

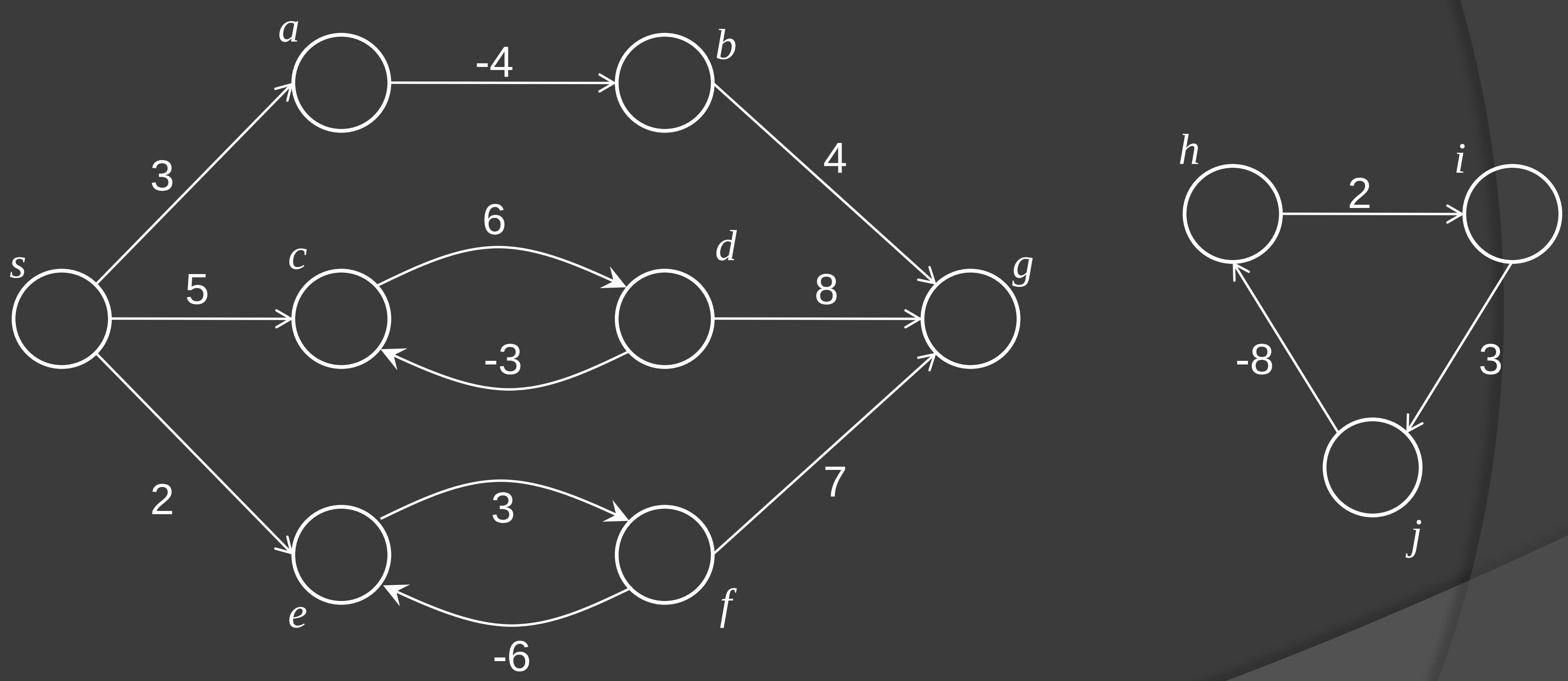
- If $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$, and
  $w(p') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$, and
  $w(p'_{ij}) < w(p_{ij})$, then
  $w(p') < w(p)$, which violates the assumption that p is a shortest path from v_1 to v_k.

Handling Negative-Weight Edges

- Edges can have negative weights.
- If the graph contains negative weights, it may or may not be a problem in computing the shortest path

Negative-Weight Edges

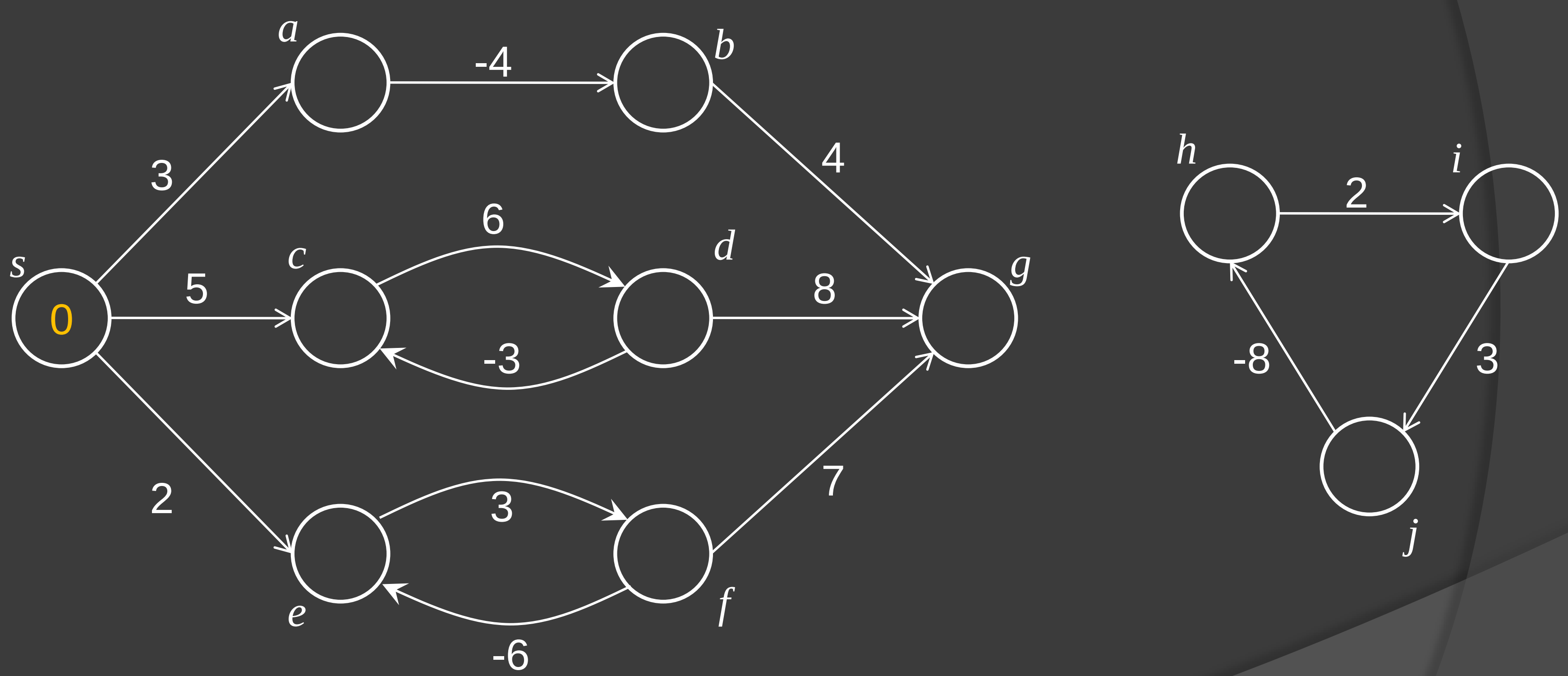
- Consider the following graph:



The shortest length path from s to s is zero (obviously)

Negative-Weight Edges (2)

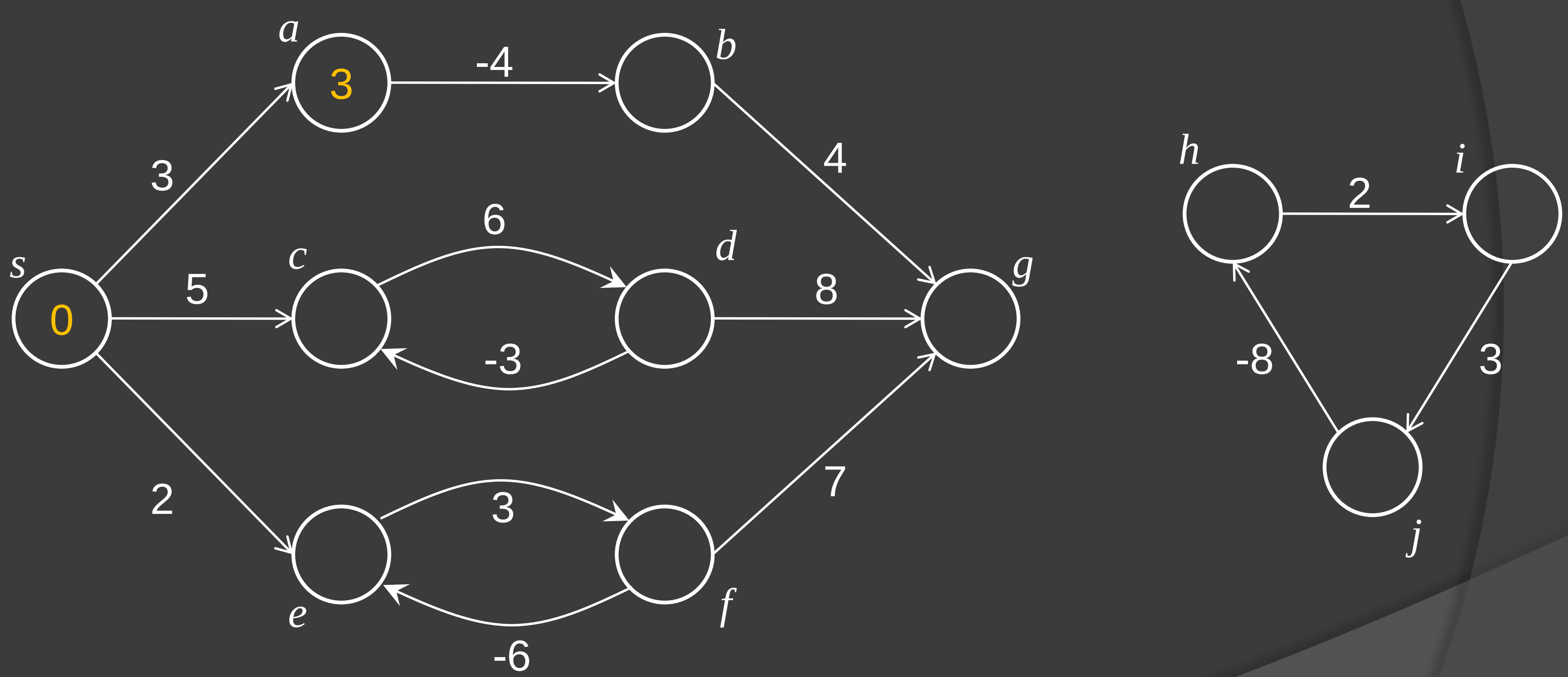
- Consider the following graph:



The shortest length path from s to a is 3

Negative-Weight Edges (3)

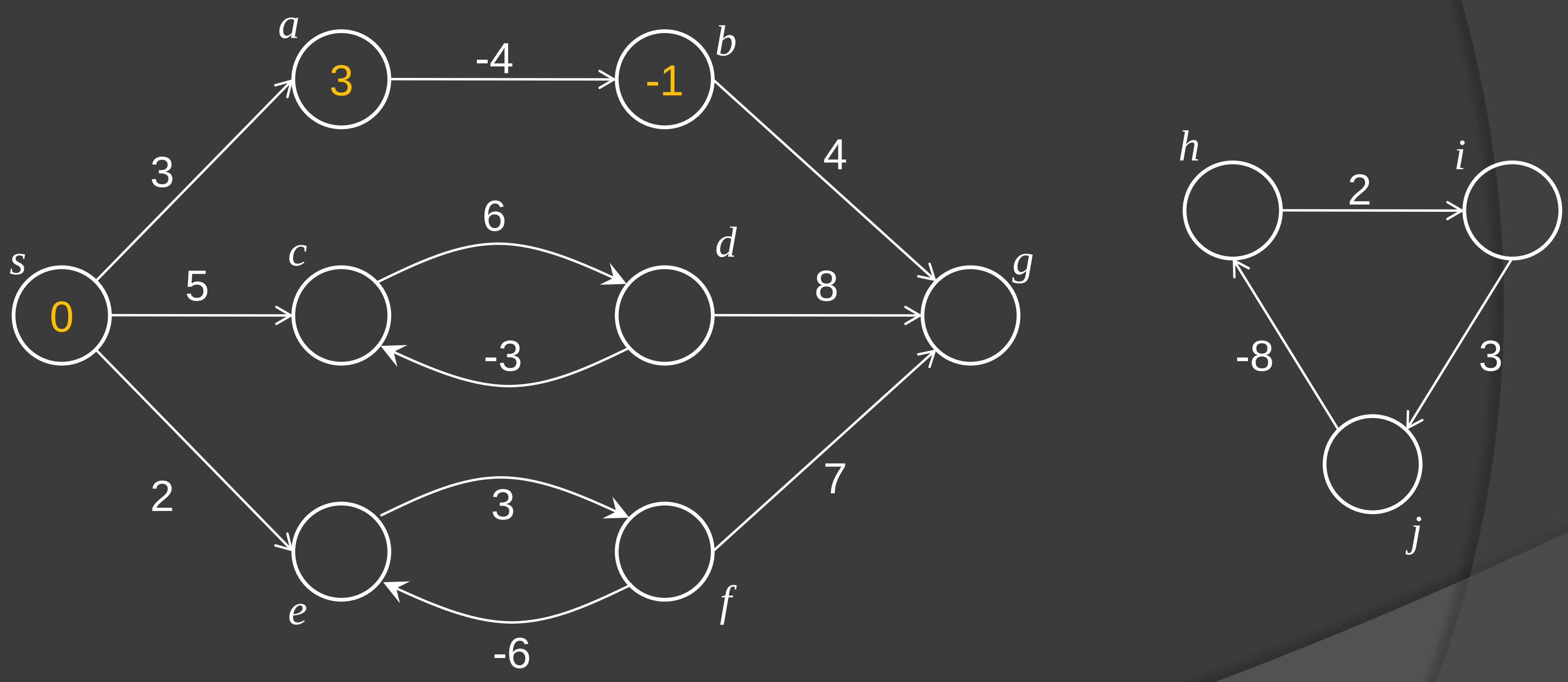
- Consider the following graph:



The shortest length path from s to b is –1

Negative-Weight Edges (4)

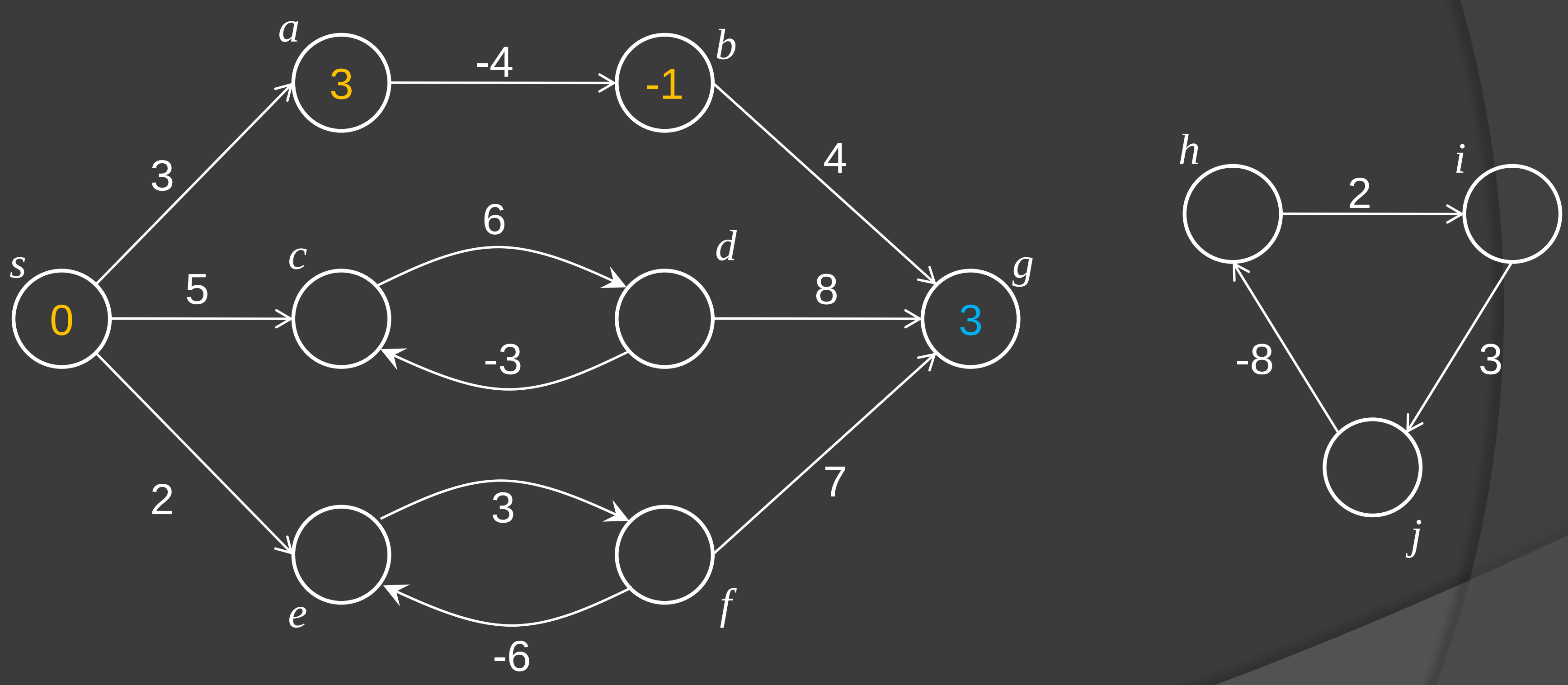
- Consider the following graph:



If we were to follow this path on to g, the shortest length path would be 3.
We'll change the color of this path length, because we're still exploring.

Negative-Weight Edges (5)

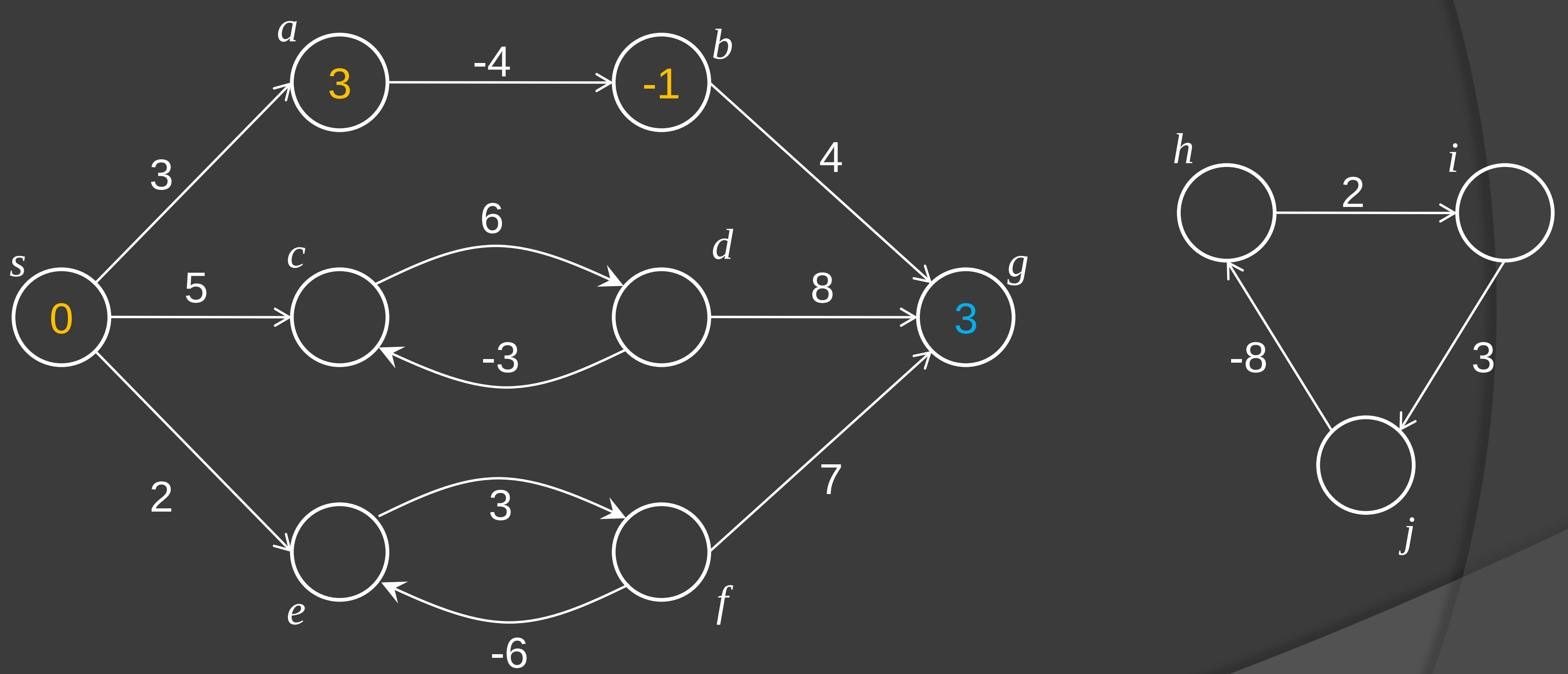
- Consider the following graph:



The fact that there's a negative-weight edge (a, b) along the way is not a problem

Negative-Weight Edges (6)

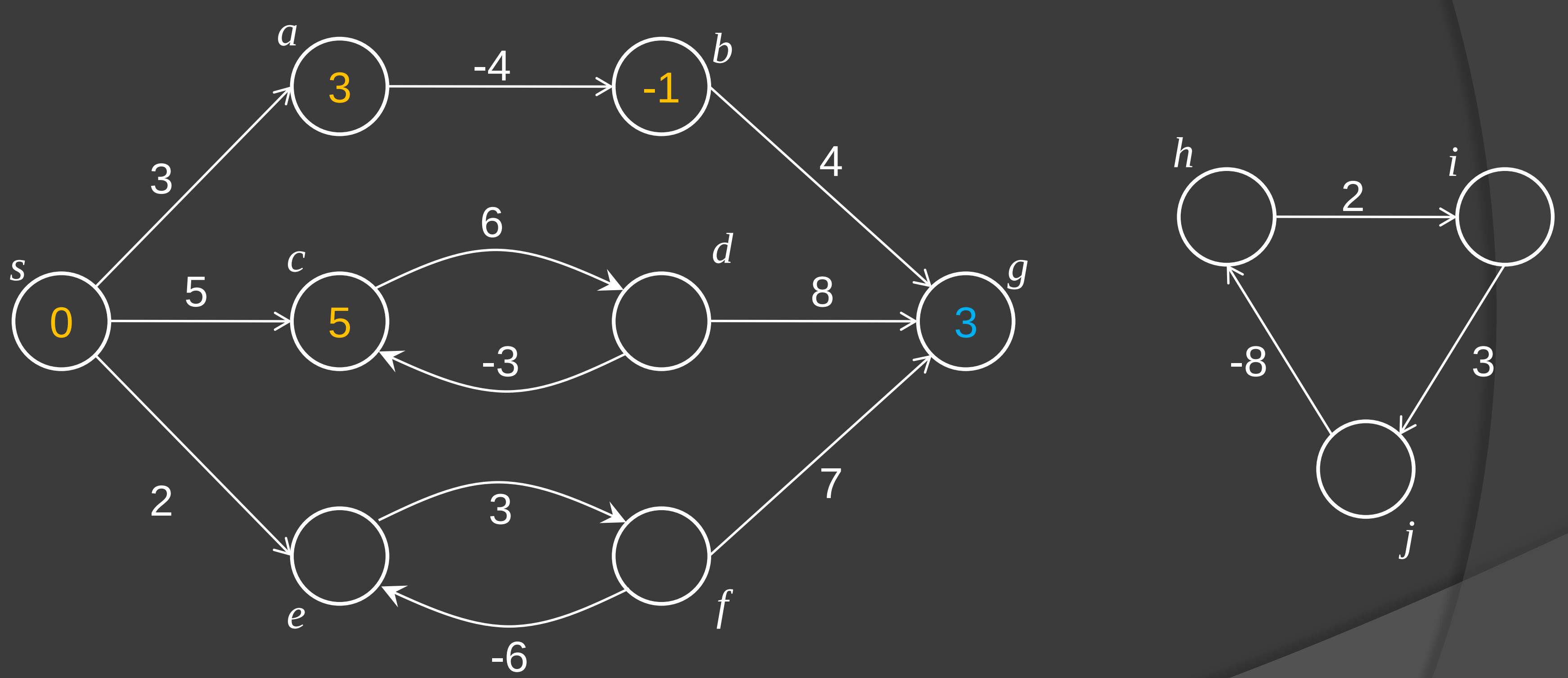
- Consider the following graph:



The shortest-length path from s to c is 5

Negative-Weight Edges (7)

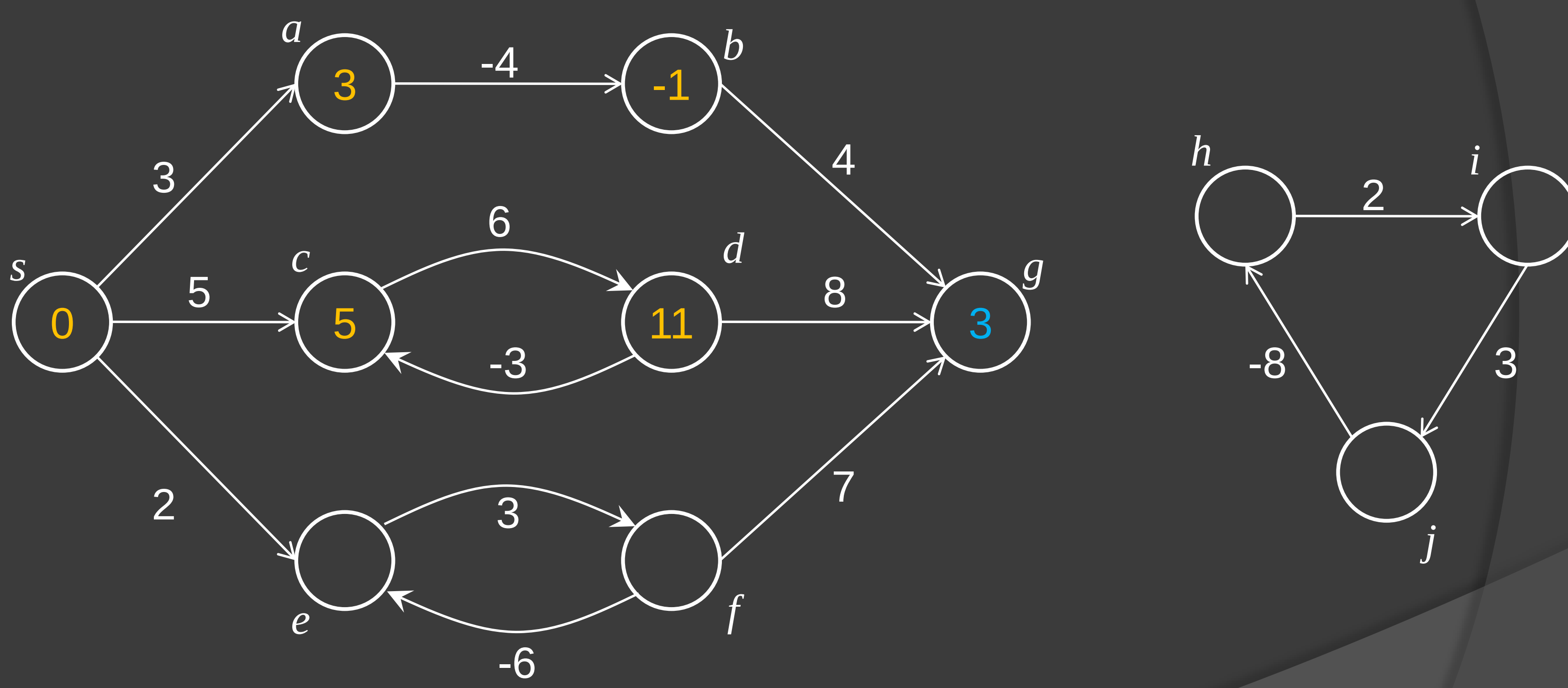
- Consider the following graph:



The shortest-length path from *s* to *d* is 11

Negative-Weight Edges (8)

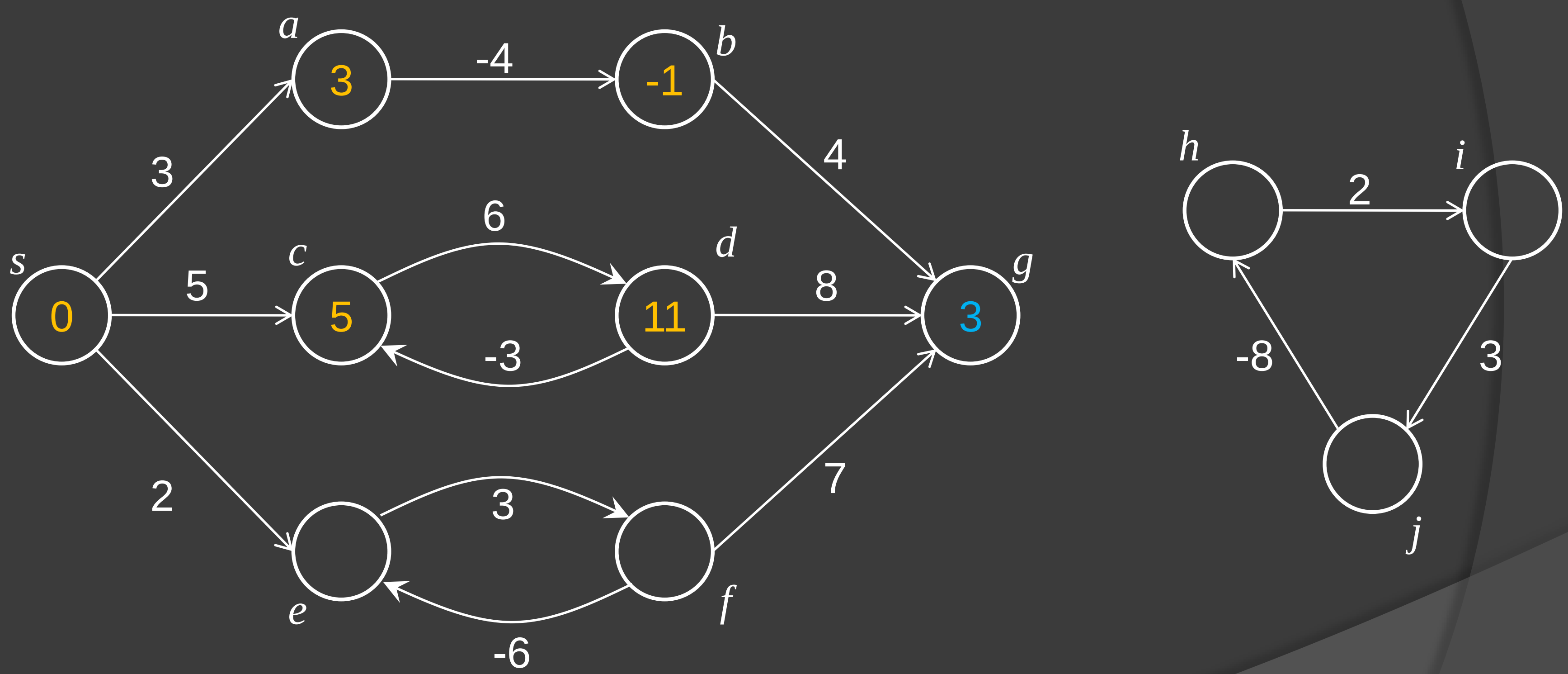
- Consider the following graph:



If we follow edge (d, c), that takes the distance from s to c to 8 via $p = \langle s, c, d, c \rangle$, with weights of $5 + 6 - 3 = 8$, which is not a new minimum, so we don't consider it. This cycle has a net weight of +3, so following the cycle only *lengthens* the path

Negative-Weight Edges (9)

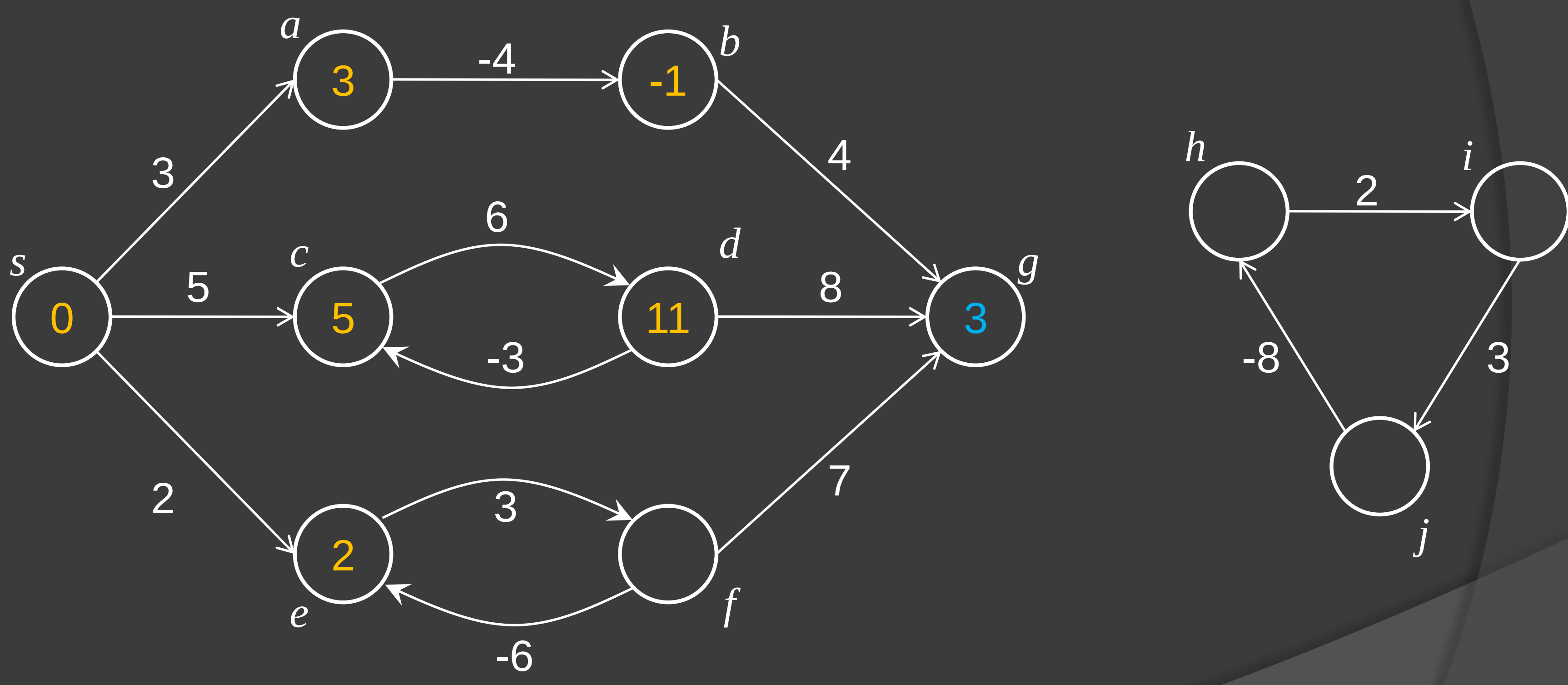
- Consider the following graph:



The shortest length path from s to e is 2.

Negative-Weight Edges (10)

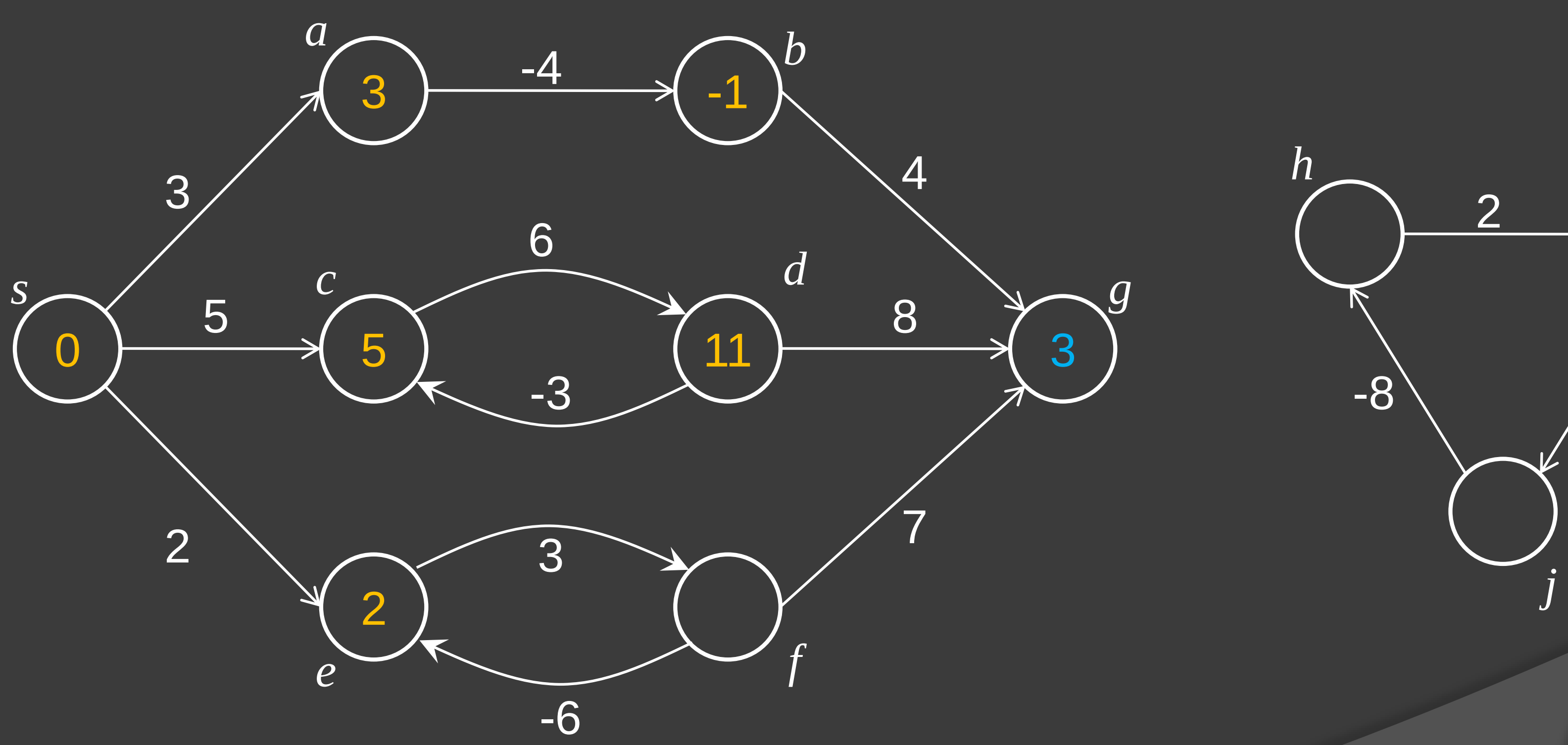
- Consider the following graph:



If we follow (s, e) and then (e, f), we have a total distance from s to f of 5. But if we follow (f, e), then the distance from s to e drops to –1. Following (e, f) and then (f, e) again will drop the distance further to –4.

Negative-Weight Edges (11)

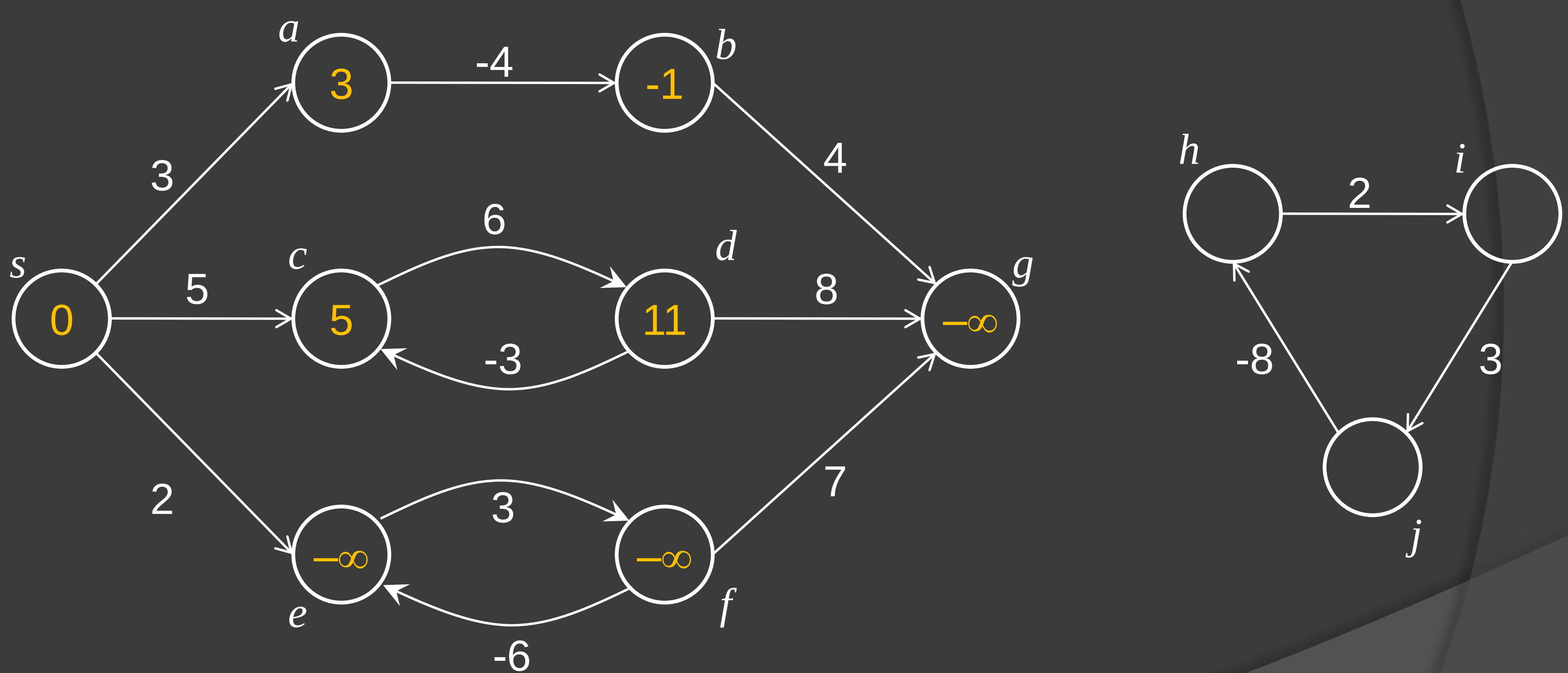
- Consider the following graph:



We're looking for the shortest-distance path, so following the cycle, which has a net weight of – 3, *does* minimize the total. But we get stuck in it, taking the total path weight to $-\infty$. Thus, the shortest-distance path from s to e, f, and g is $-\infty$.

Negative-Weight Edges (12)

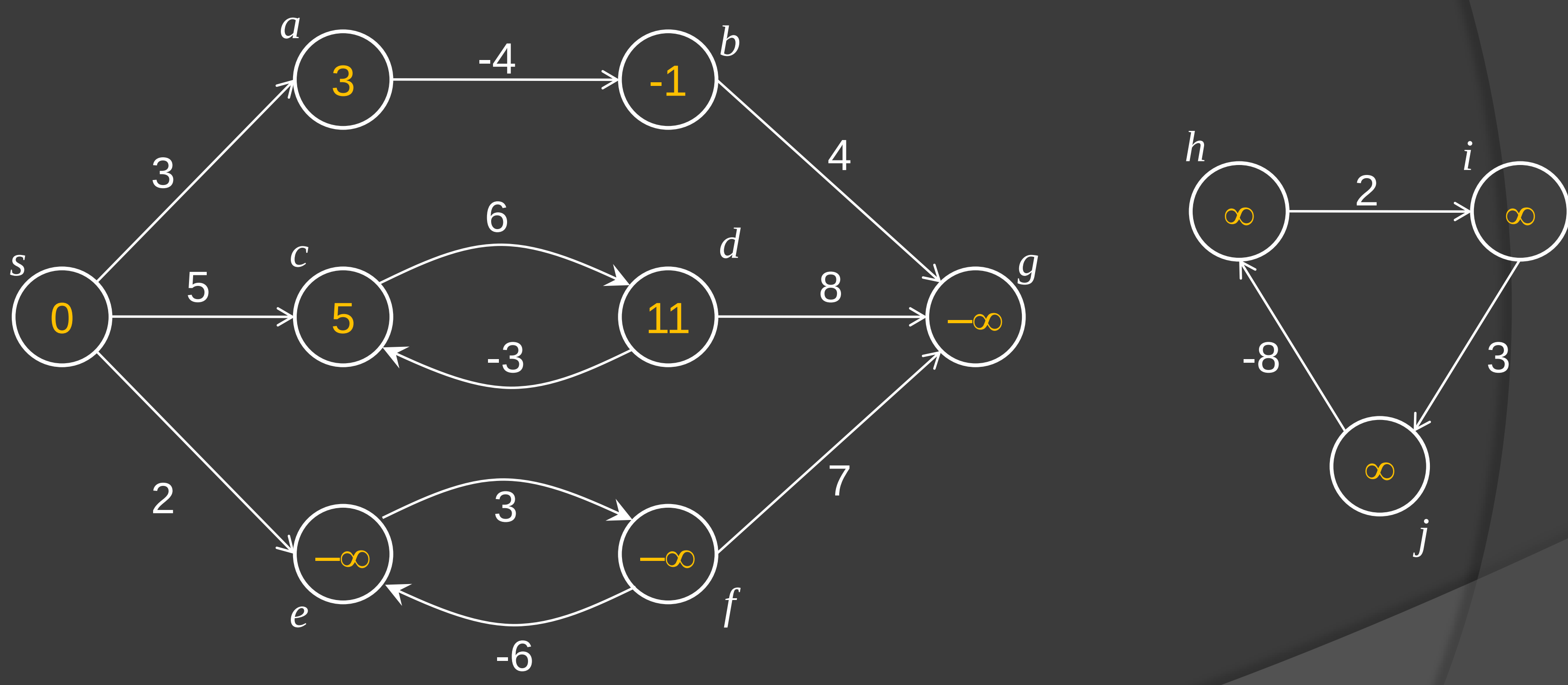
- Consider the following graph:



Despite the fact that $\{h, i, j\}$ contains a negative-weight cycle (net weight: –3), the fact that nodes h, i, and j are all unreachable from s means that the distance from s to each of $\{h, i, j\}$ is ∞

Negative-Weight Edges (13)

- Consider the following graph:



Despite the fact that $\{h, i, j\}$ contains a negative-weight cycle (net weight: –3), the fact that nodes h, i, and j are all unreachable from s means that the distance from s to each of $\{h, i, j\}$ is ∞

Cycles

- Clearly, reachable, negative-weight cycles can’t be part of a shortest-length path
- Similarly, positive-weight cycles would not be taken, as following the cycle would only raise the path’s length above the minimum.
- Zero-weight cycles can be removed without changing the net path length, so they effectively don’t raise or lower the total.
- Therefore, we restrict our shortest-path discussion to paths with no cycles, making the maximum path length $|V| - 1$

Negative Cycles

- Some algorithms work only if there are no negative-weight edges in the graph (at all, whether they're reachable or not)
  - We'll be clear when they're allowed and not allowed.

Representing Shortest Paths

- We may want to know the path (list of vertices, in order), in addition to its length
- For each vertex $v \in V$, we maintain a predecessor value, $v.\pi$, that is either the vertex before v along the path, or NIL.
- To get our resulting path, therefore, we start at the destination, and work backwards towards the source, using the predecessor values.

More on the Predecessors

- The predecessors don't necessarily result in a straight-line path from the source to the destination (it may branch).
- Rather, they form a shortest-path ***tree***, just as we formed trees of reachable nodes in DFS.
- We define V_π, as the set of vertices with non-NIL predecessors, plus the source vertex s:

$$V_\pi = \{v \in V\colon \pi \neq \text{NIL}\} \cup s$$

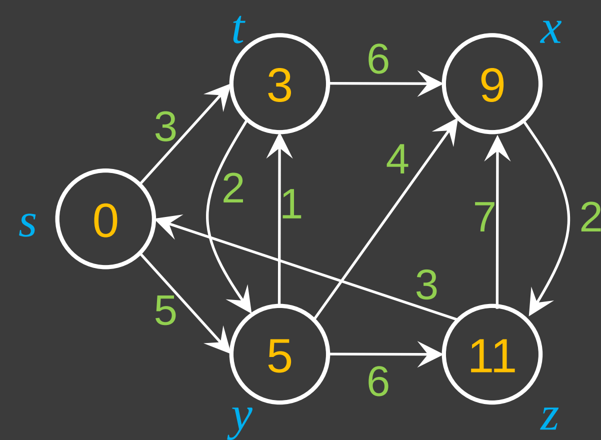
- The predecessors also create a set of edges:

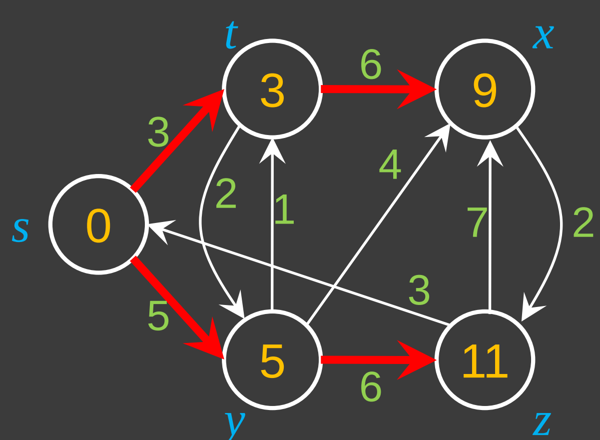
$$E_\pi = \{(v.\pi, v) \in E\colon v \in V_\pi - \{s\}\}$$

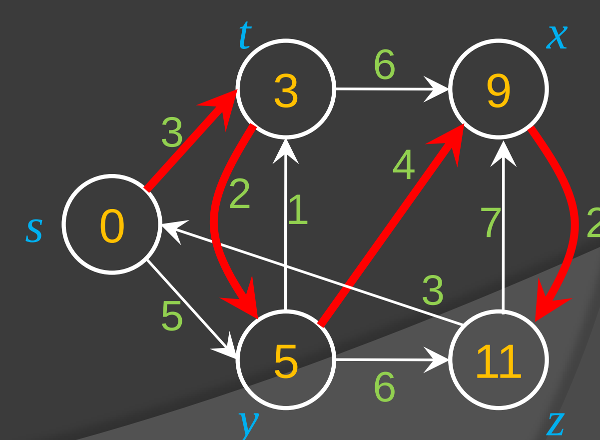
More on the Predecessors (2)

- Since the predecessors define a set of vertices and a set of edges, they also define a graph: $G_\pi = (V_\pi, E_\pi)$
- G_π is a shortest-path tree, rooted at s, that embodies a shortest path from s to every node reachable from s.
- Note that we said, "*a* shortest-path tree," and not "*the* shortest-path tree". There IS exactly one shortest-path *length*, but there may be multiple trees with that length.

Shortest Path Trees







Relaxation

- Our shortest-path algorithms use ***relaxation***, a process of successively tightening the upper bound (i.e., lowering the upper bound).
  - See the footnote on p.648 for an explanation of the somewhat counter-intuitive name for this process.
  - In general, refers to letting a solution, temporarily, violate a constraint, and trying to fix these violations.
- Relaxation maintains an upper-bound (worst-case) distance $v.d$ from the source to every other vertex (i.e., from s to $v \in \{V - \{s\}\}$

About that “Upper Bound”

- Because our shortest path will, at worst, cover all of the $|V| - 1$ edges in G, we could start with an upper bound of the sum of the weights of all of the edges in the graph
  - This could run into problems with negative path weights and unreachable vertices.
  - Since we’re looking for an upper bound on what the worst-case path length could possibly be (and we’ll lower that upper bound as we learn we can), we can start with something even larger, like ∞

Relaxation

- We start our shortest-path algorithm by setting this upper bound to zero for s, and to ∞ for all other vertices
  - We don't even know whether or not the other vertices are reachable yet, so the upper bound (worst-case distance) to them *is* ∞
  - While we're initializing, we set every vertex's predecessor to NIL.

INITIALIZE-SINGLE-SOURCE Algorithm

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$

2 $\quad v.d = \infty$ // shortest (known) path to all else

3 $\quad v.\pi = \text{NIL}$ // no path → no predecessor

4 $s.d = 0$ // distance from s to s is 0

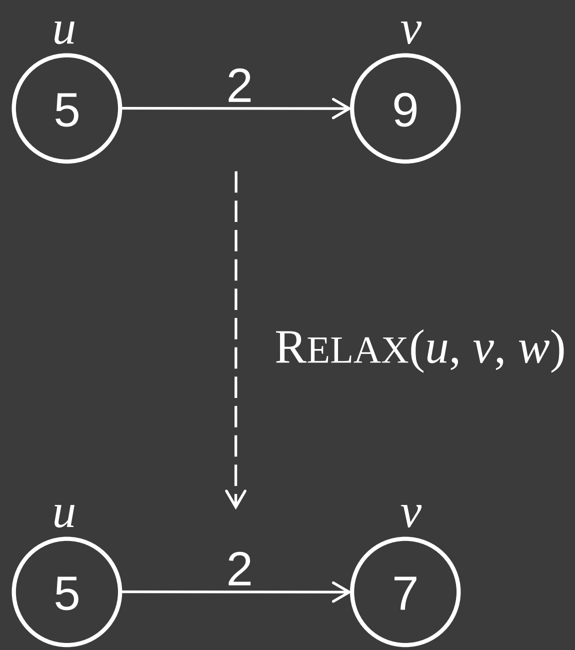
This algorithm pretty obviously runs in $\Theta(V)$ time

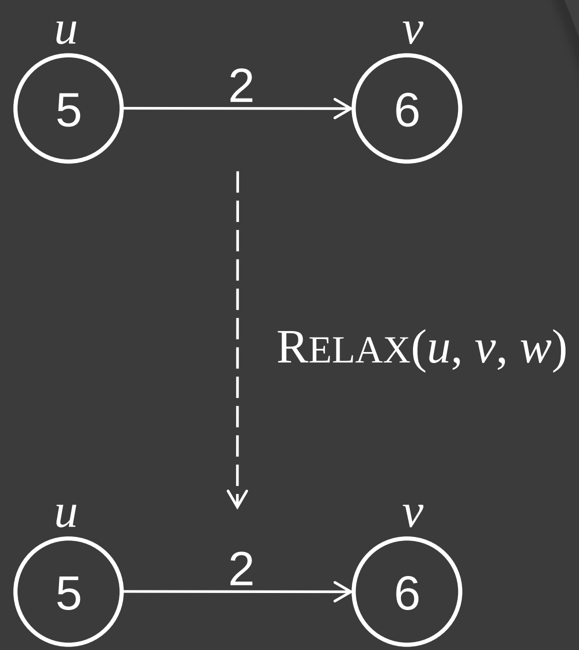
Relaxation

- Relaxing an edge (u, v):
  - See if we can lower the upper-bound (i.e., have we found a shorter path than what we've seen so far) to v by going through u
  - If so, we have a new minimum: update $v.d$ and $v.\pi$

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Relaxation – Two Examples





Next Steps

- All of our shortest-path algorithms start with INITIALIZE-SINGLE-SOURCE and then use RELAX to systematically relax edges.
- They differ in the number of times they relax an edge, and the sequence in which they work
  - In the BELLMAN-FORD algorithm, edges can be relaxed many times
  - DIJKSTRA's algorithm and the dag algorithm relax each edge exactly once

Properties of Shortest Paths & Relaxation

- Triangle inequality (Lemma 24.10)
  - For any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- Upper-Bound property (Lemma 24.11)
  - We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes
- No-Path property (Corollary 24.12)
  - If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$

Properties (2)

- Convergence Property (Lemma 24.14)
  - If $s \leadsto u \to v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.
- Path-relaxation property (Lemma 24.15)
  - If $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and the edges of p are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with the relaxations of the edges in p

Properties (3)

- Predecessor-subgraph property (Lemma 24.17)
  - Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

BELLMAN-FORD Algorithm

- Solves the single-source shortest-paths problem in the general case
- Given a weighted, directed graph $G = (V, E)$, with source s and weight function $w: E \to R$, the algorithm searches the edges, relaxing them as it goes
- Works with negative-weight edges
- Negative-weight cycles are detected. If one is found, the algorithm returns FALSE, and there is no solution; otherwise it returns TRUE

BELLMAN-FORD Algorithm

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i = 1$ to $|V[G]| - 1$
3 **for** each edge $(u, v) \in E[G]$
4 RELAX(u, v, w)
5 **for** each edge $(u, v) \in E[G]$
6 **if** $v.d > u.d + w(u, v)$
7 **return** FALSE
8 **return** TRUE

Lines 2-4: Relax all of the edges $|V[G]| - 1$ times

Note: edges are not considered in any particular (specified) order!

Lines 5-7: Search for negative-weight cycles

BELLMAN-FORD Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |V[G]| – 1
3        for each edge (u, v) ∈ E[G]
4              RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6        if v.d > u.d + w(u, v)
7              return FALSE
8  return TRUE
```

Run Time: Initialization: $\Theta(V)$

Lines 2 – 4: $\Theta(VE)$ (V passes on E edges)

Lines 5 – 7: $\Theta(E)$

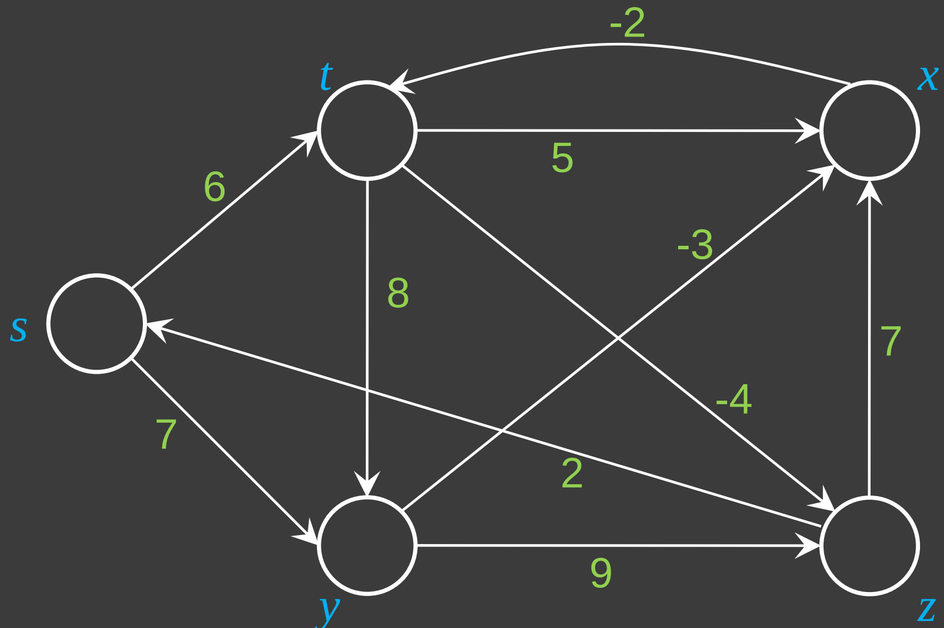
Total: $\Theta(VE)$

Bellman-Ford Algorithm

Bellman-Ford *may* find the shortest path after a single pass through the loop in lines 2-4, but it has no way of knowing whether it has or not, so it will *always* run $|V| - 1$ passes on all $|E|$ edges.

The text has a proof of Bellman-Ford's correctness (pp. 653-654)

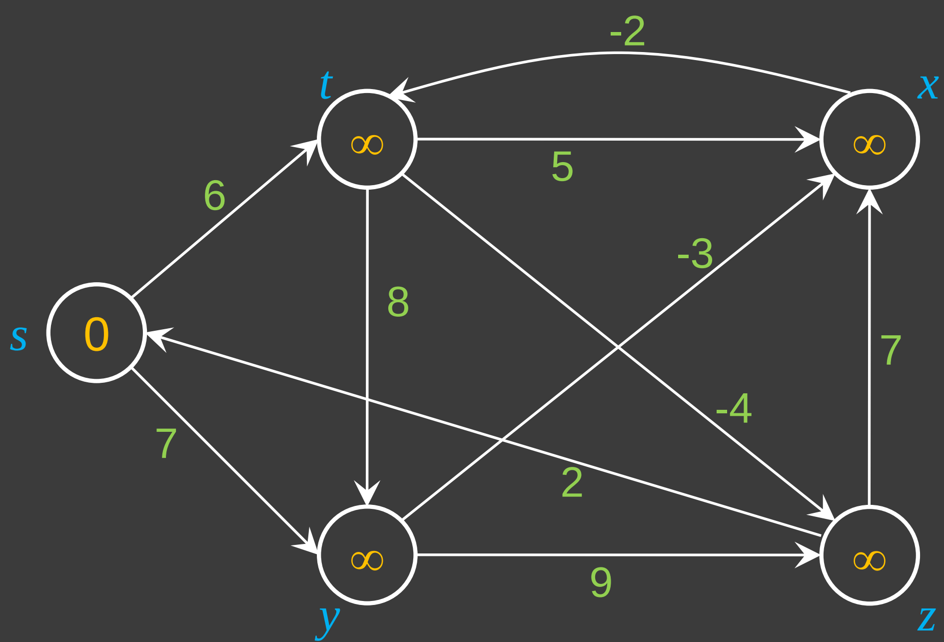
BELLMAN-FORD Walkthrough



We'll show the *v.d* values inside each vertex

First, call INITIALIZE-SINGLE-SOURCE

Bellman-Ford Walkthrough

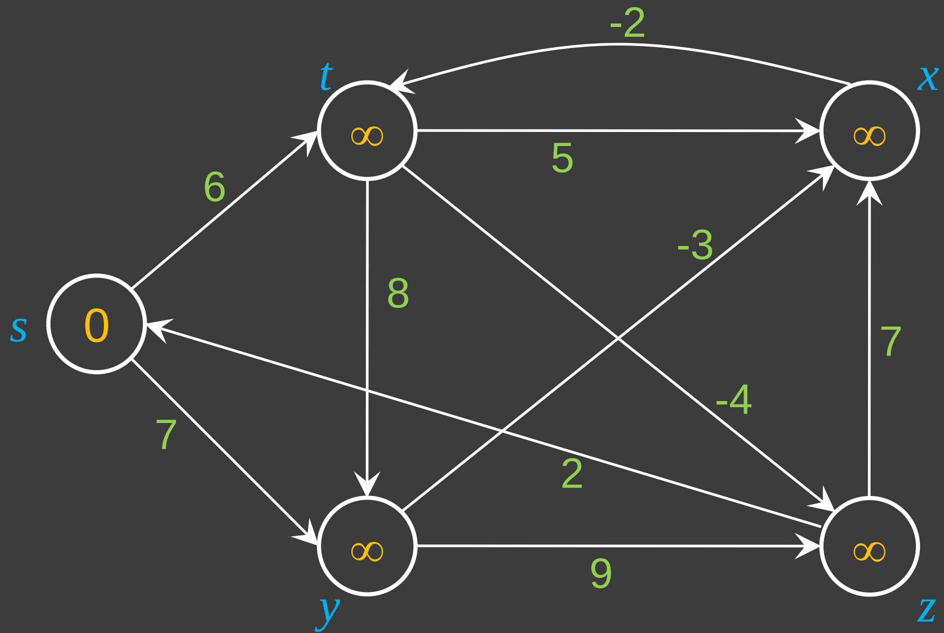


Relax edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(t, x): We reduce $x.d$ if $x.d > t.d + w(t, x)$, but it isn't – no change
(t, y): We reduce $y.d$ if $y.d > t.d + w(t, y)$, but it isn't – no change
(t, z), (x, t), (y, x), (y, z), (z, x), (z, s): Same thing.

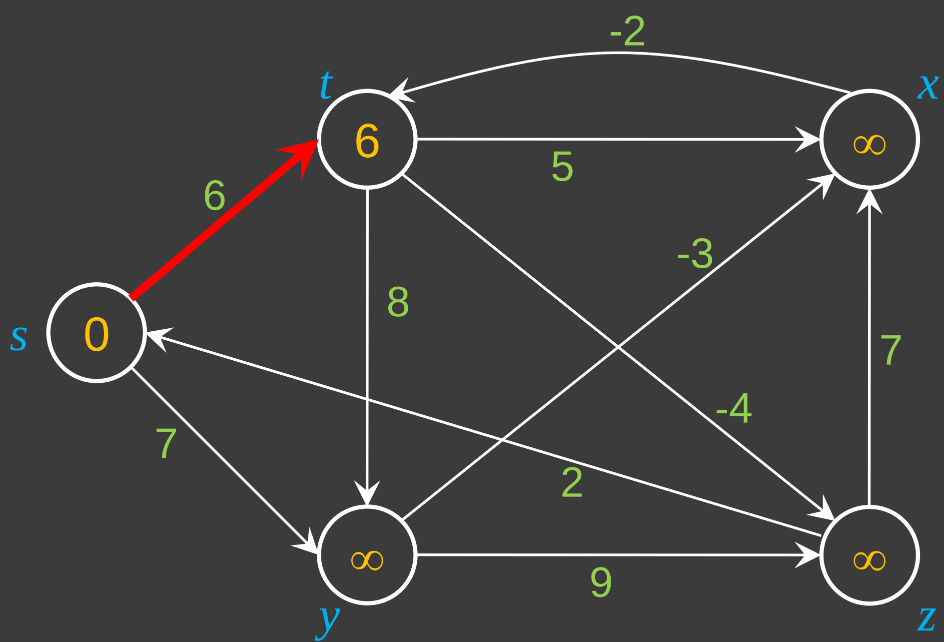
BELLMAN-FORD Walkthrough



RELAX edges in this order:
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(s, t): We reduce $t.d$ if $t.d > s.d + w(s, t)$: $\infty > 0 + 6$, so $t.d = (0 + 6)$
Also, $t.\pi = s$

BELLMAN-FORD Walkthrough

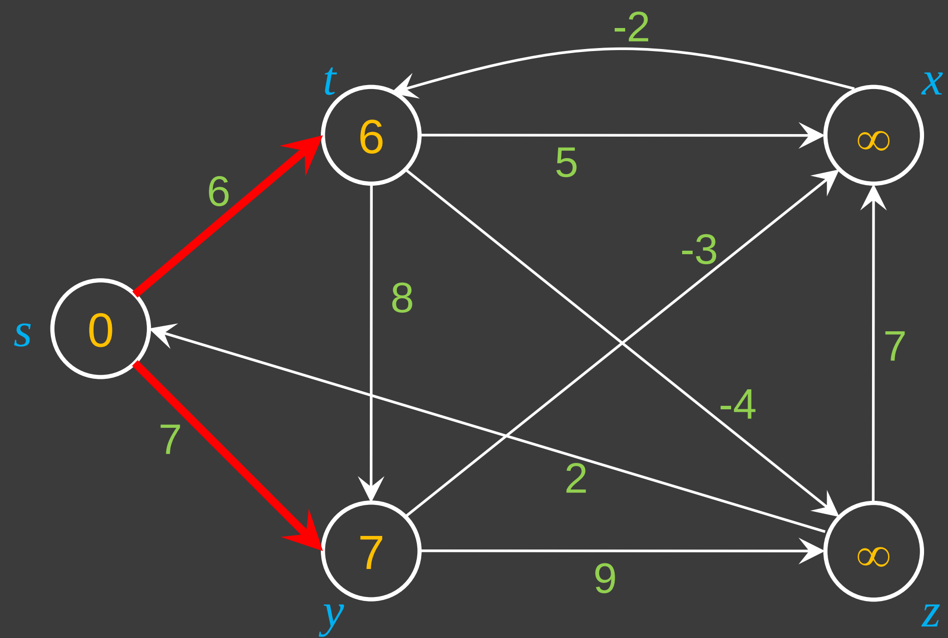


RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(s, y): We reduce $y.d$ if $y.d > s.d + w(s, y)$: $\infty > 0 + 7$, so $y.d = 7$. Also, $y.\pi = s$

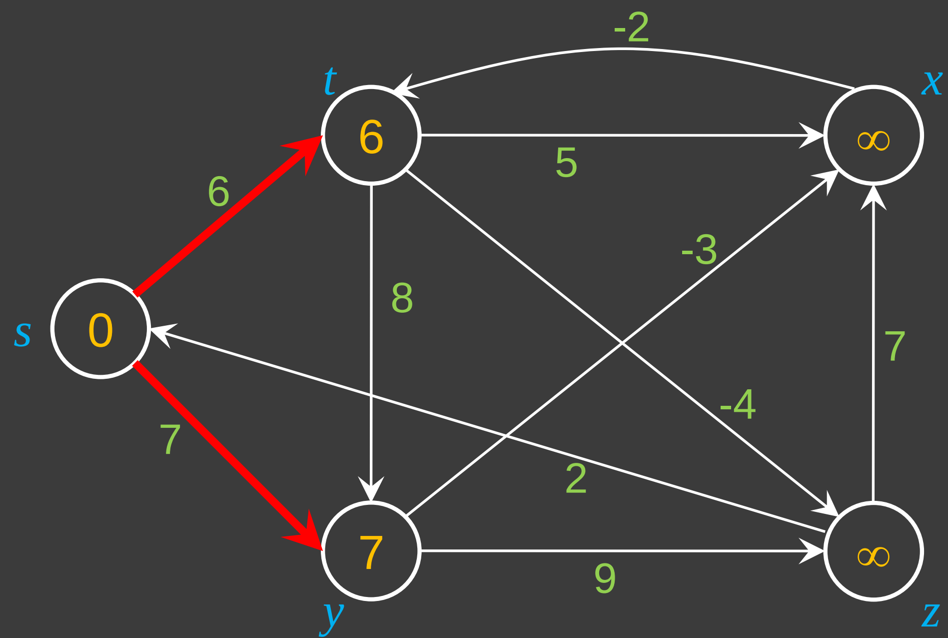
BELLMAN-FORD Walkthrough



RELAX edges in this order:
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

That's the end of the first pass through all of the edges. We will eventually make $|V|$-1 passes. Since there are 5 vertices, we will make 4 passes. One down, three to go.

BELLMAN-FORD Walkthrough

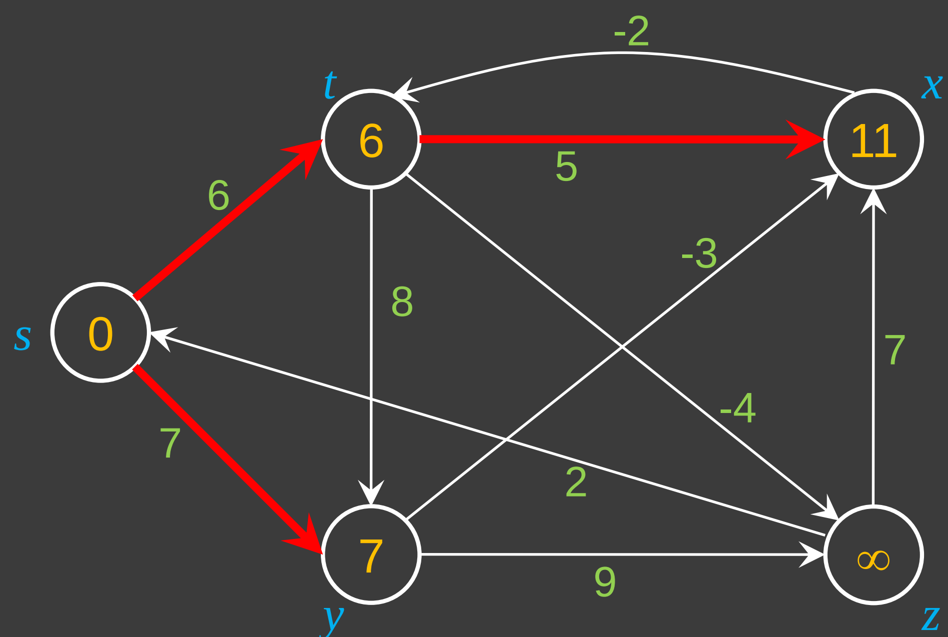


RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(t, x): We reduce $x.d$ if $x.d > t.d + w(t, x)$: ∞ ***is*** $> 6 + 5$, so $x.d = 11$; $x.\pi = t$

BELLMAN-FORD Walkthrough

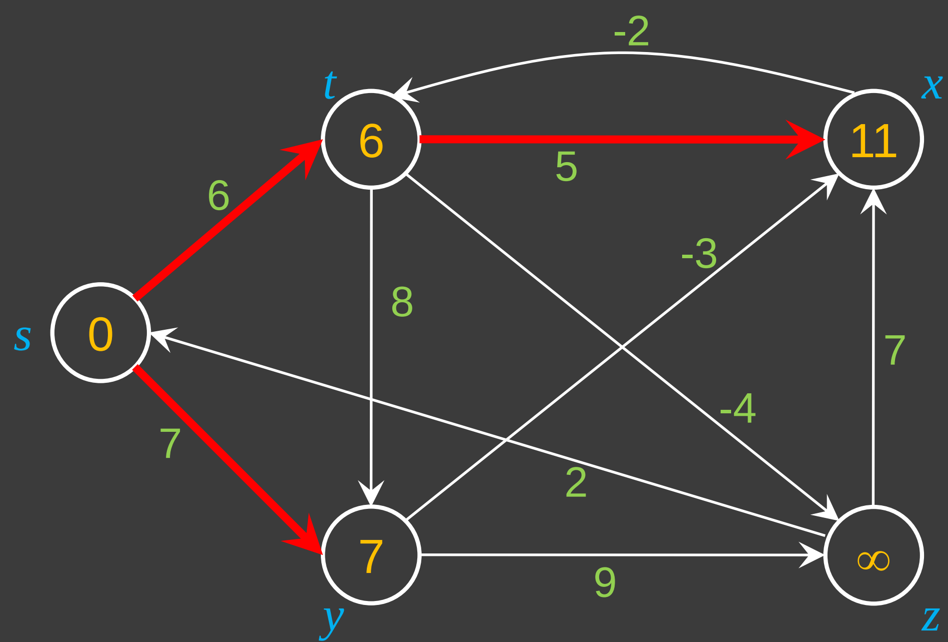


RELAX edges in this order:

(*t*, *x*)
(*t*, *y*)
(*t*, *z*)
(*x*, *t*)
(*y*, *x*)
(*y*, *z*)
(*z*, *x*)
(*z*, *s*)
(*s*, *t*)
(*s*, *y*)

(*t*, *y*): We update *y.d* if *y.d* > *t.d* + *w*(*t*, *y*): 7 ***is not*** > 6 + 8, so keep going

BELLMAN-FORD Walkthrough

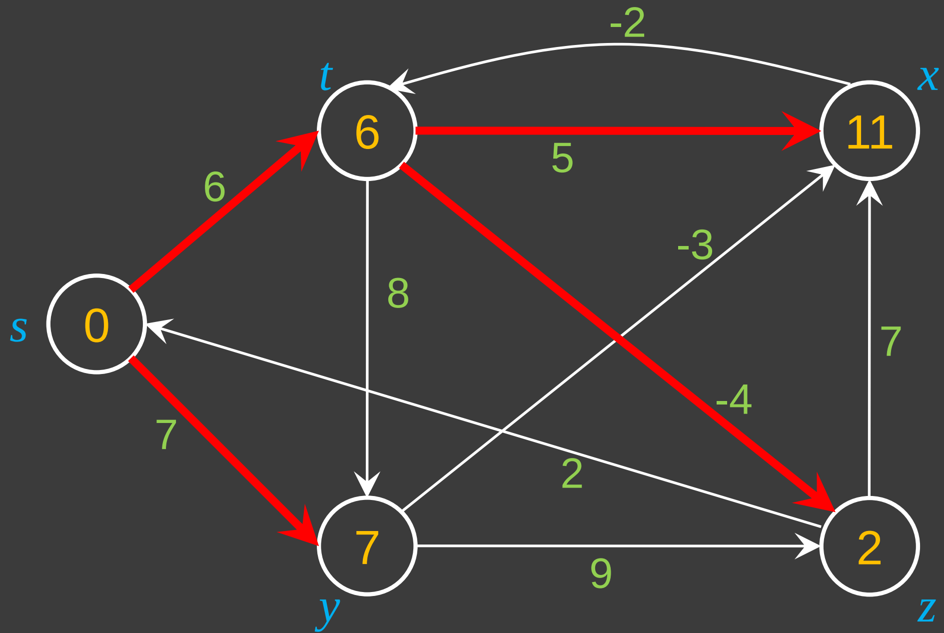


RELAX edges in this order:

(*t*, *x*)
(*t*, *y*)
(*t*, *z*)
(*x*, *t*)
(*y*, *x*)
(*y*, *z*)
(*z*, *x*)
(*z*, *s*)
(*s*, *t*)
(*s*, *y*)

(t, z): We update $z.d$ if $z.d > t.d + w(t, z)$: ∞ ***is*** $> 6 + (-4)$, so $z.d = 2$; $z.\pi = t$

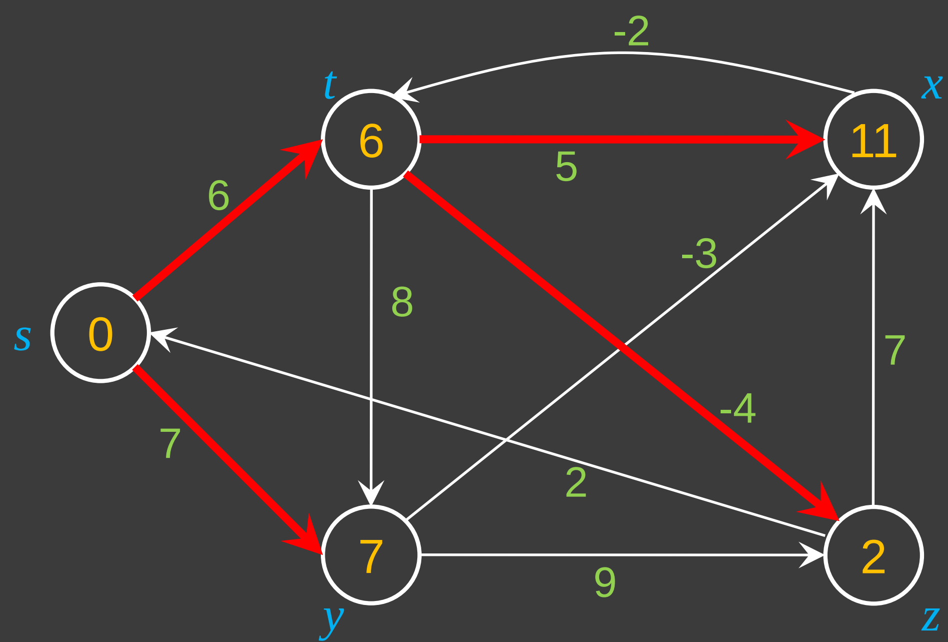
BELLMAN-FORD Walkthrough



RELAX edges in this order:
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(x, t): We update $t.d$ if $t.d > x.d + w(x, t)$: 6 ***is not*** > 11 + (–2), so keep going

BELLMAN-FORD Walkthrough



RELAX edges in this order:

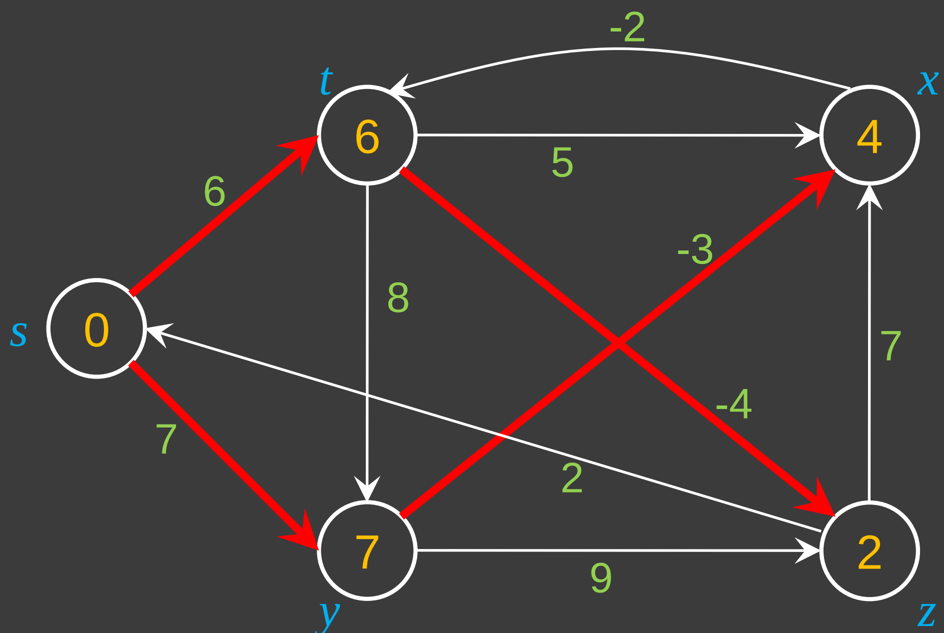
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(y, x): We update $x.d$ if $x.d > y.d + w(y, x)$: 11 ***is*** > 7 + (–3), so set $x.d = 4$

We *had* $x.\pi$ as t, but we do better getting to x via y, so x's predecessor becomes y

$x.\pi$ = y

BELLMAN-FORD Walkthrough



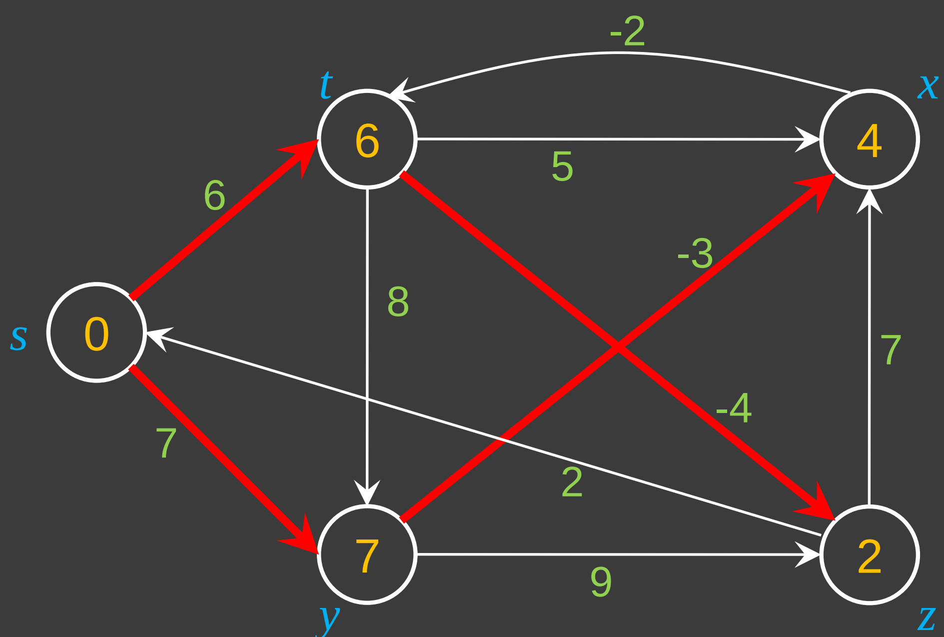
RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(y, z): We update $z.d$ if $z.d > y.d + w(y, z)$: 2 ***is not*** > 7 + 9, so keep going

(z, x): We update $x.d$ if $x.d > z.d + w(z, x)$: 4 ***is not*** > 2 + 7, so keep going

Bellman-Ford Walkthrough

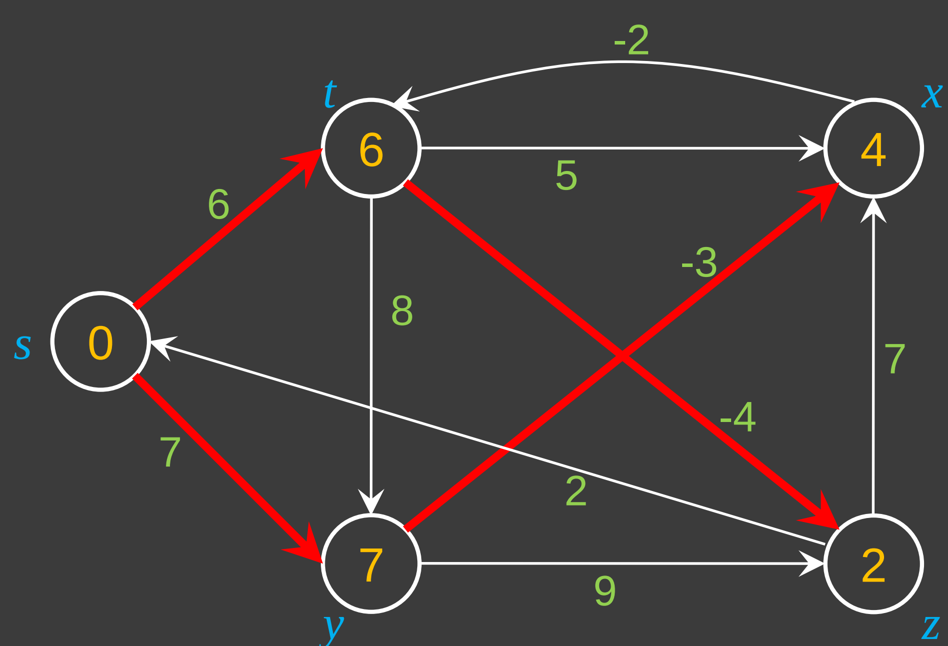


Relax edges in this order:

(*t*, *x*)
(*t*, *y*)
(*t*, *z*)
(*x*, *t*)
(*y*, *x*)
(*y*, *z*)
(*z*, *x*)
(*z*, *s*)
(*s*, *t*)
(*s*, *y*)

(*z*, *s*): We update *s.d* if *s.d* > *z.d* + *w*(*z*, *s*): 0 ***is not*** > 2 + 2, so keep going
(*s*, *t*): We update *t.d* if *t.d* > *s.d* + *w*(*s*, *t*): 6 ***is not*** > 0 + 6, so keep going
(*s*, *y*): We update *y.d* if *y.d* > *s.d* + *w*(*s*, *y*): 7 ***is not*** > 0 + 7, so keep going
That's the last edge, so that's the end of the second pass. Two passes down, 2 to go.

Bellman-Ford Walkthrough

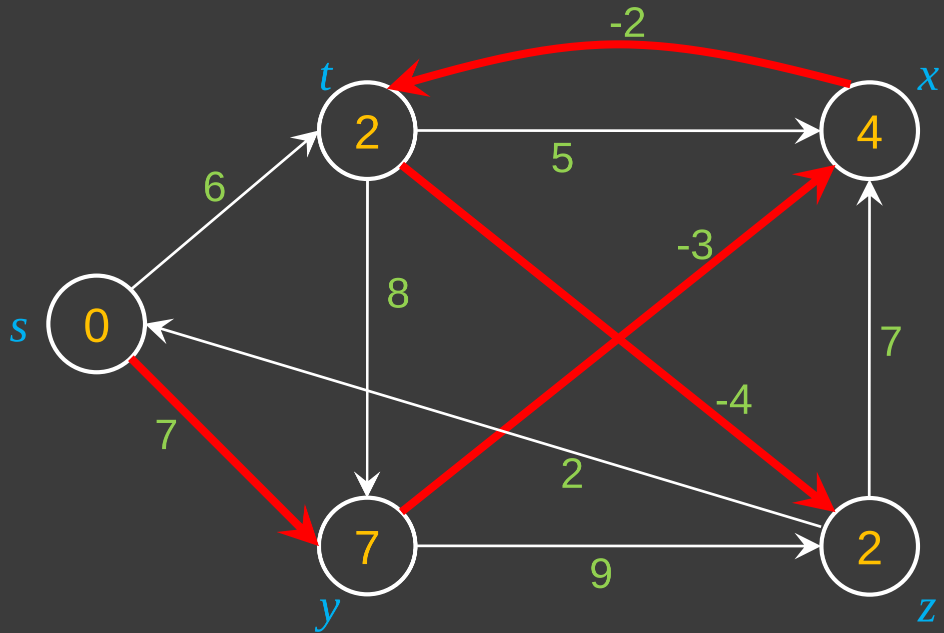


Relax edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(t, x): We update $x.d$ if $x.d > t.d + w(t, x)$: 4 ***is not*** > 6 + 5, so keep going
(t, y): We update $y.d$ if $y.d > t.d + w(t, y)$: 7 ***is not*** > 6 + 8, so keep going
(t, z): We update $z.d$ if $z.d > t.d + w(t, z)$: 2 ***is not*** > 6 + (–4), so keep going
(x, t): We update $t.d$ if $t.d > x.d + w(x, t)$: 6 ***is*** > 4 + (–2), so set $t.d = 2$;
t's predecessor was s, but we can do better getting to t via x, so $t.\pi = x$ (instead of s)

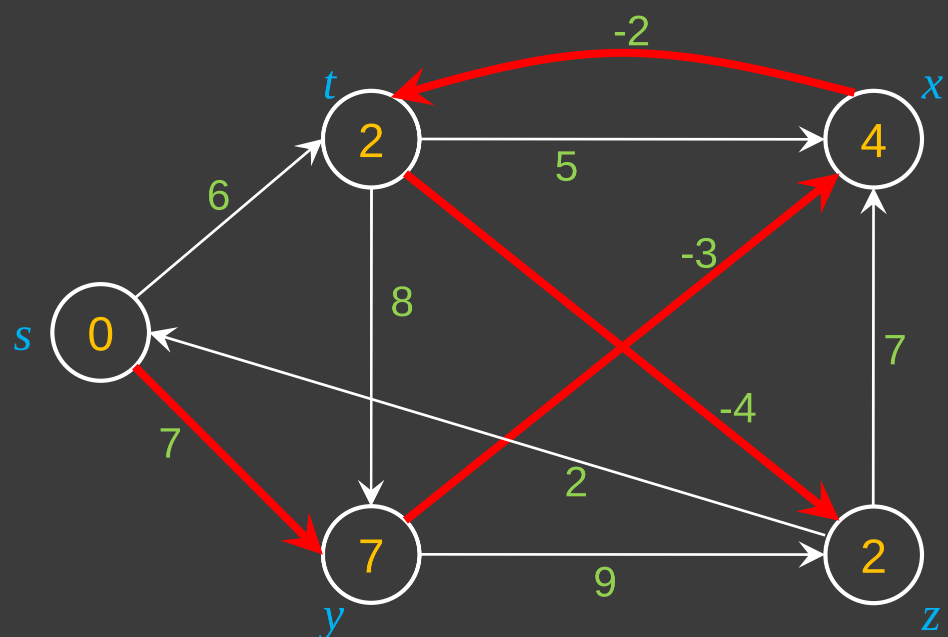
BELLMAN-FORD Walkthrough



RELAX edges in this order:
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(y, x): We update $x.d$ if $x.d > y.d + w(y, x)$: 4 ***is not*** > 7 + (–3), so keep going
(y, z): We update $z.d$ if $z.d > y.d + w(y, z)$: 2 ***is not*** > 7 + 9, so keep going
(z, x): We update $x.d$ if $x.d > z.d + w(z, x)$: 4 ***is not*** > 2 + 7, so keep going
(z, s): We update $s.d$ if $s.d > z.d + w(z, s)$: 0 ***is not*** > 2 + 2, so keep going

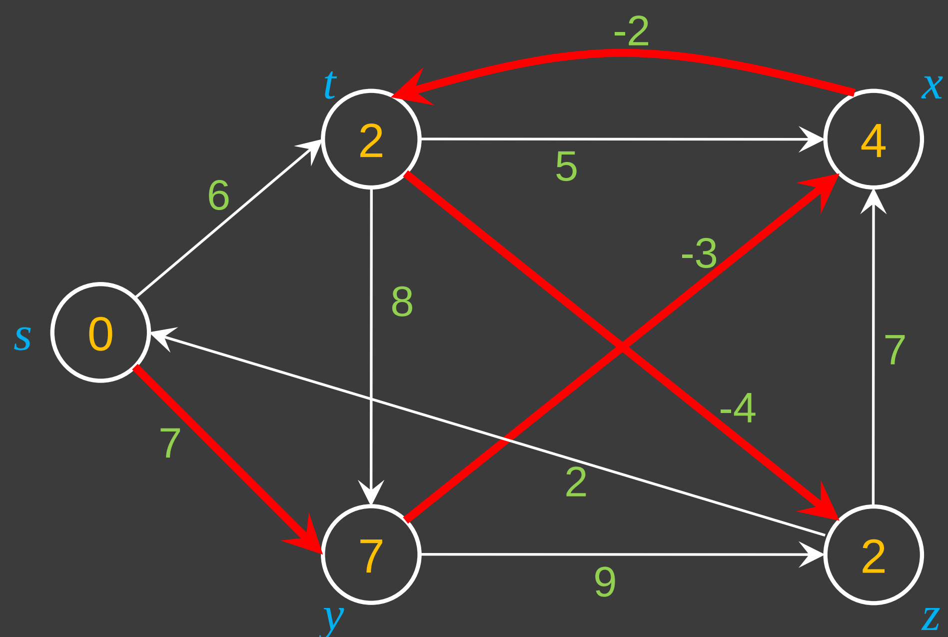
BELLMAN-FORD Walkthrough



RELAX edges in this order:
(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(s, t): We update $t.d$ if $t.d > s.d + w(s, t)$: 2 ***is not*** > 0 + 6, so keep going
(s, y): We update $y.d$ if $y.d > s.d + w(s, y)$: 7 ***is not*** > 0 + 7, so keep going
That's the last edge. That's the end of the third pass. Three passes down, one to go

BELLMAN-FORD Walkthrough



RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

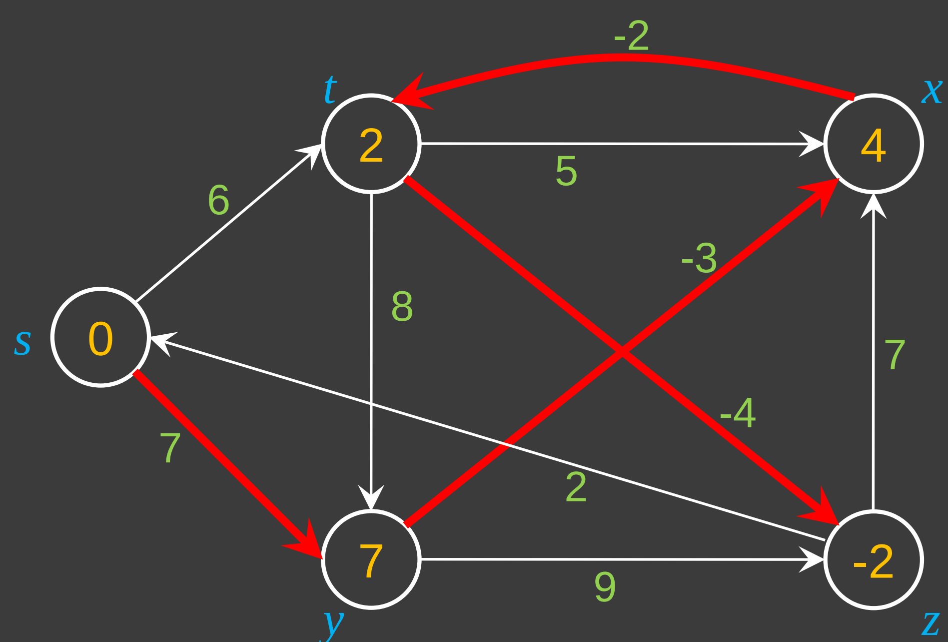
(t, x): We update $x.d$ if $x.d > t.d + w(t, x)$: 4 ***is not*** $> 2 + 5$, so keep going

(t, y): We update $y.d$ if $y.d > t.d + w(t, y)$: 7 ***is not*** $> 2 + 8$, so keep going

(t, z): We update $z.d$ if $z.d > t.d + w(t, z)$: 2 ***is*** $> 2 + (-4)$, so set $z.d = (-2)$

$z.\pi$ was already t; we've just realized that the new path length was even better than what we had, so we (again) set $z.\pi = t$, even though that's what it already is

BELLMAN-FORD Walkthrough



RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(x, t): We update $t.d$ if $t.d > x.d + w(x, t)$: 2 ***is not*** > 4 + (–2), so keep going

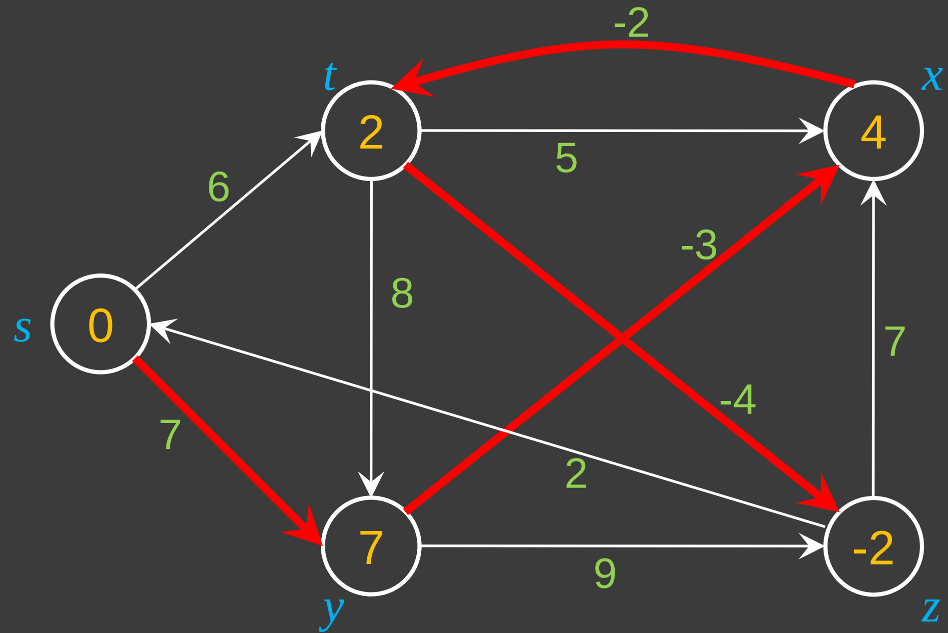
(y, x): We update $x.d$ if $x.d > y.d + w(y, x)$: 4 ***is not*** > 7 + (–3), so keep going

(y, z): We update $z.d$ if $z.d > y.d + w(y, z)$: –2 ***is not*** > 7 + 9, so keep going

(z, x): We update $x.d$ if $x.d > z.d + w(z, x)$: 4 ***is not*** > (–2) + 7, so keep going

(z, s): We update $s.d$ if $s.d > z.d + w(z, s)$: 0 ***is not*** > (–2) + 2, so keep going

BELLMAN-FORD Walkthrough

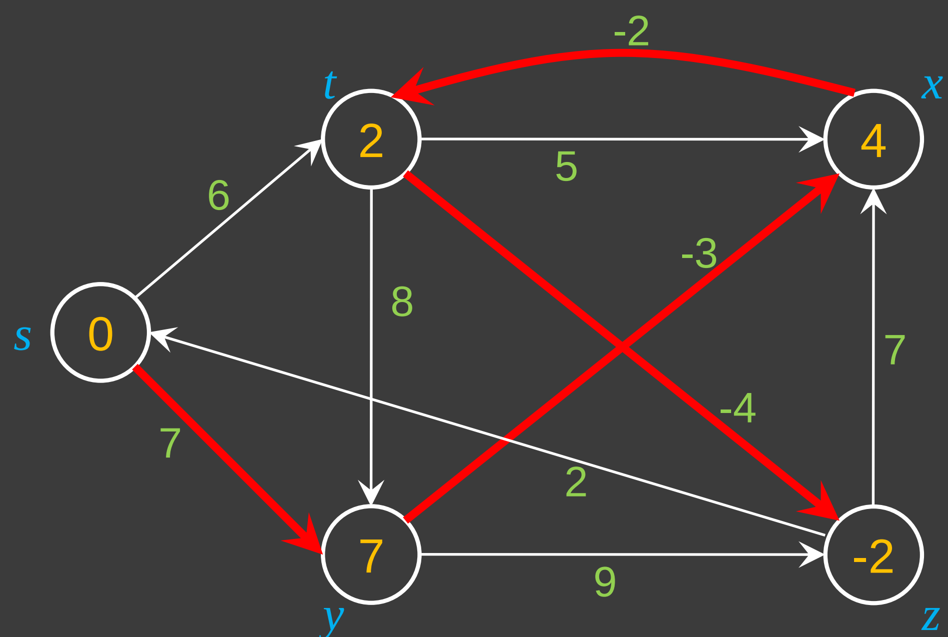


RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

(s, t): We update $t.d$ if $t.d > s.d + w(s, t)$: 2 ***is not*** > 0 + 6, so keep going
(s, y): We update $y.d$ if $y.d > s.d + w(s, y)$: 7 ***is not*** > 0 + 7, so keep going
That's the end of the fourth (and final pass)!

BELLMAN-FORD Walkthrough

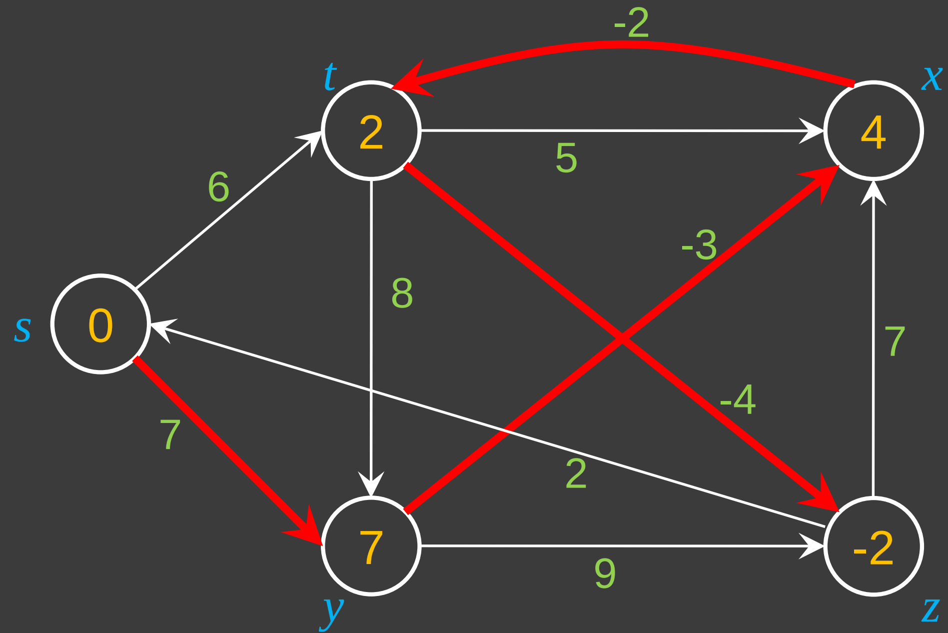


RELAX edges in this order:

(t, x)
(t, y)
(t, z)
(x, t)
(y, x)
(y, z)
(z, x)
(z, s)
(s, t)
(s, y)

One more thing to do – check every edge (u, v) to see if $v.d > u.d + w(u, v)$
(Lines 5-7 of the algorithm)

Bellman-Ford Walkthrough



Edge (u, v)	$d[u]$	$w(u, v)$	$d[u] + w(u, v)$	$d[v]$
(t, x)	2	5	7	4
(t, y)	2	8	10	7
(t, z)	2	-4	-2	-2
(x, t)	4	-2	2	2
(y, x)	7	-3	4	4
(y, z)	7	9	16	-2
(z, x)	-2	7	5	4
(z, s)	-2	2	0	0
(s, t)	0	6	6	2
(s, y)	0	7	7	7

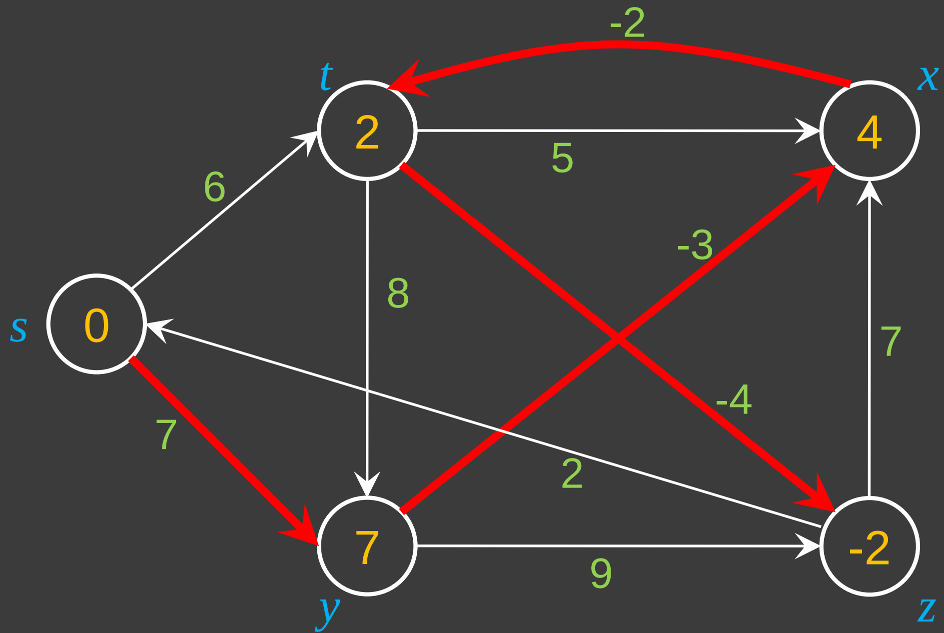
Check every edge (u, v) to see if $v.d > u.d + w(u, v)$

Nothing in this column…

…is < its counterpart in this column

So, **return** TRUE – we have a solution (and no negative cycles)!

Bellman-Ford Walkthrough



So, the shortest-length path from s to…

…s is 0: $p = \langle\rangle$
…t is 2: $p = \langle s, y, x, t\rangle$
…x is 4: $p = \langle s, y, x\rangle$
…y is 7: $p = \langle s, y\rangle$
…z is -2: $p = \langle s, y, x, t, z\rangle$

Speeding up BELLMAN-FORD

- If the graph is acyclic, we don't have to worry about negative cycles (if there are NO cycles, there can't be any negative-weight cycles!)
- In the case of a dag, we can use the Topological-Sort (§22.4) to reduce the single-source shortest path solution time from $\Theta(VE)$ to $\Theta(V+E)$

DAG-SHORTEST-PATHS Algorithm

DAG-SHORTEST-PATHS(G, w, s)
1 Topologically sort the vertices of G (see §22.4)
2 INITIALIZE-SINGLE-SOURCE(G, s)
3 **for** each vertex u, taken in topologically sorted order
4   **for** each vertex $v \in Adj[u]$
5   RELAX(u, v, w)

Run time: Line 1: $\Theta(V+E)$ (see 22.4)
Line 2: $\Theta(V)$
Line 5: RELAX will run $|E|$ times, $\Theta(1)$ per
Total: $\Theta(V+E)$

DIJKSTRA's Algorithm

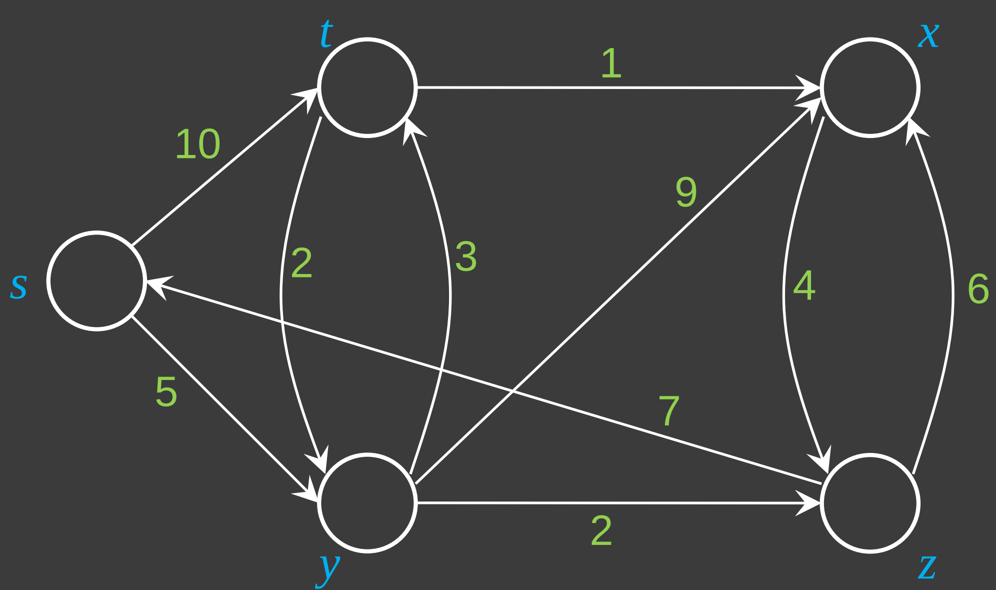
- Edge weights must be ≥ 0
- Essentially a weighted version of BFS
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights ($v.d$)
- We maintain two sets of vertices:
  - S: vertices whose final shortest-path weights have been determined, and
  - Q: a priority queue containing $\{V - S\}$

DIJKSTRA's Algorithm

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$ // no vertices have been finalized
$Q = G.V$ // Q: a Min-priority queue, with $V.d$'s
while $Q \neq \emptyset$ // until the queue is empty
    u = EXTRACT-MIN(Q) // get the first queue item
    $S = S \cup \{u\}$ // add u to the finished vertex set
    for each vertex $v \in Adj[u]$
        RELAX(u, v, w)

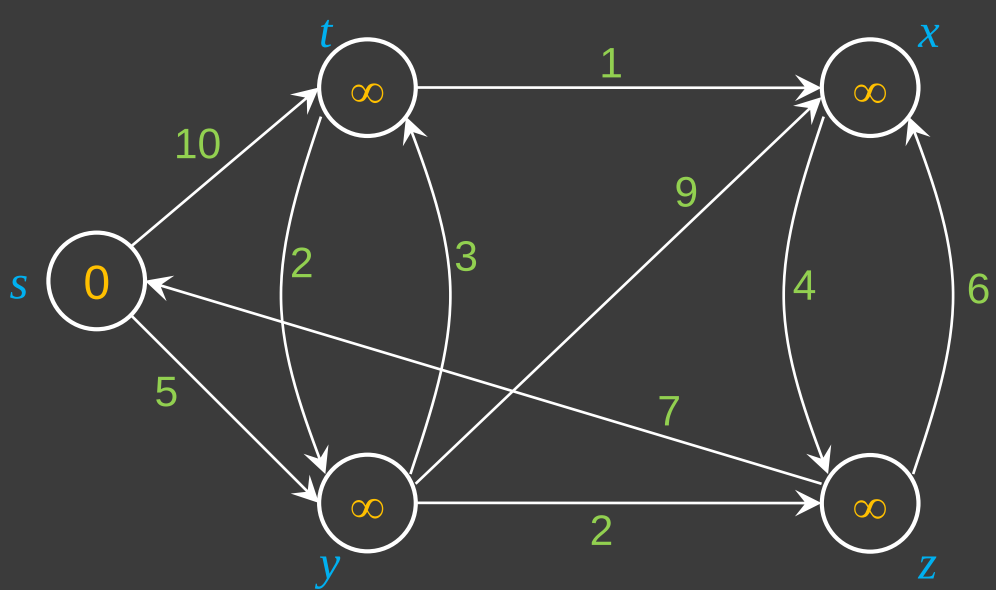
DIJKSTRA Walkthrough



We'll show the *v.d* values inside each vertex

Start by calling INITIALIZE-SINGLE-SOURCE

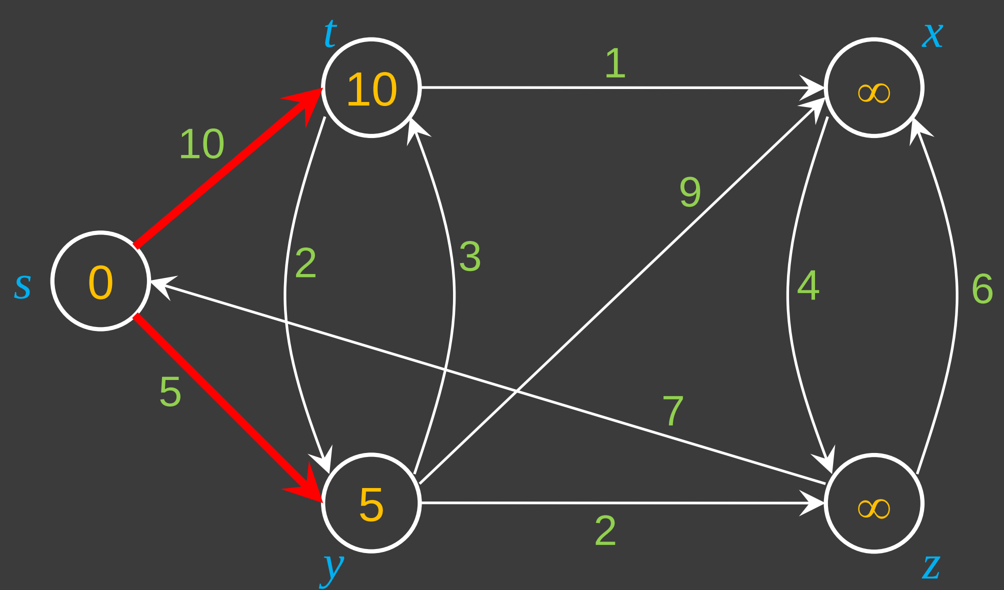
DIJKSTRA Walkthrough



$Q = \{s(0), t(\infty), x(\infty), y(\infty), z(\infty)\}$
$S = \{\}$

We're starting at vertex s, so it was initialized as having a distance of zero, putting it first in the min-queue Q. All other vertices follow s, tied at ∞
We extract the vertex s from the front of Q, and add it to S. Then we relax all of the edges leading from s, $\{(s, t)$ and $(s, y)\}$, updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough

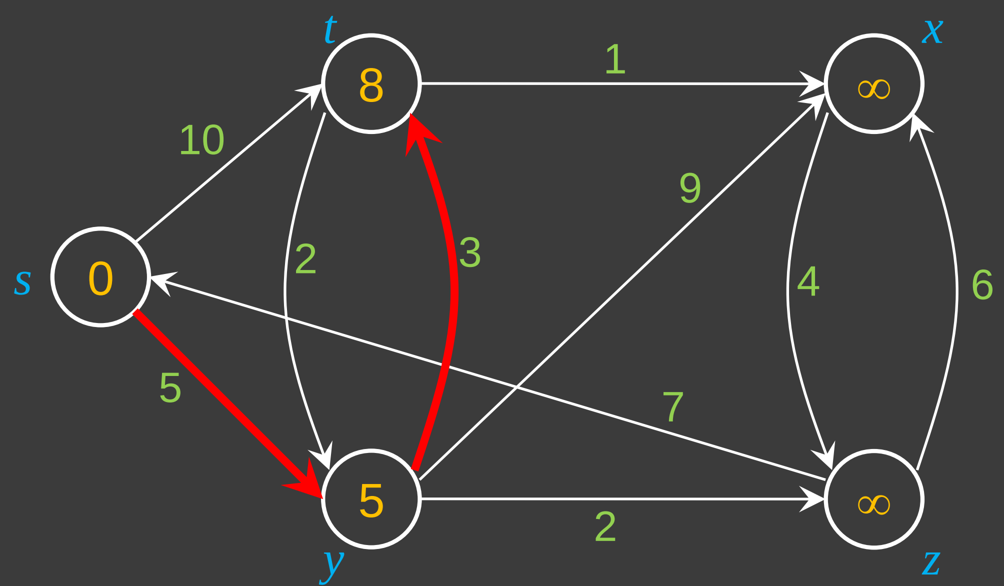


$Q = \{y(5), t(10), x(\infty), z(\infty)\}$
$S = \{s\}$

Relaxing (y, t) gives us a new $t.d$ of 8 and a new $t.\pi$ of y

We extract the vertex y from the front of Q, and add it to S. Then we relax all of the edges leading from y, $\{(y, t), (y, z), (y, x)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough



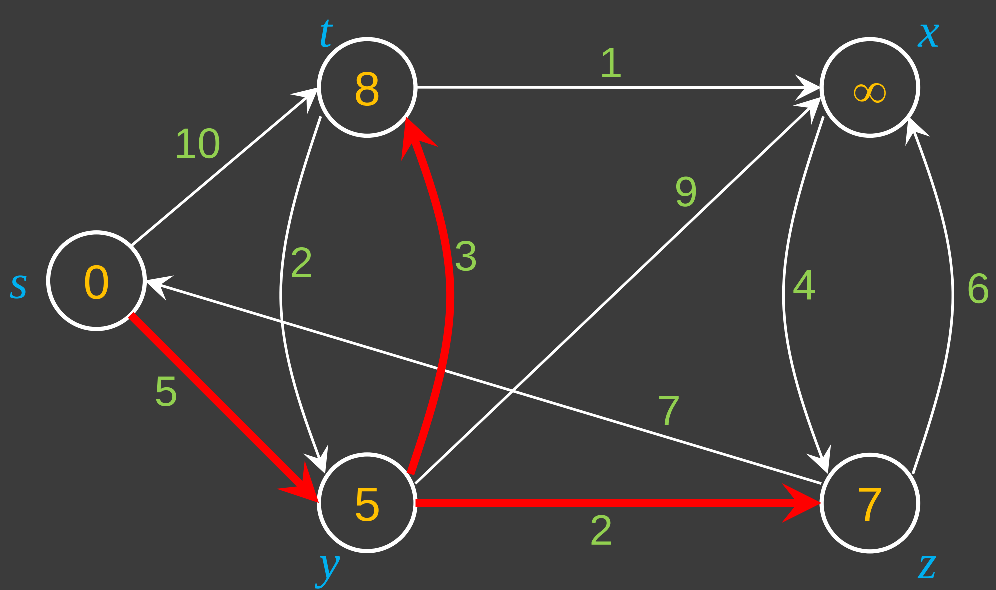
$Q = \{t(8), x(\infty), z(\infty)\}$
$S = \{s, y\}$

Relaxing (y, t) gives us a new $t.d$ of 8 and a new $t.\pi$ of y

Relaxing (y, z) gives us a new $z.d$ of 7 and a new $t.\pi$ of y

We extract the vertex v from the front of Q, and add it to S. Then we relax all of the edges leading from v, $\{(y, t), (y, z), (y, x)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough



$Q = \{z(7), t(8), x(\infty)\}$
$S = \{s, y\}$

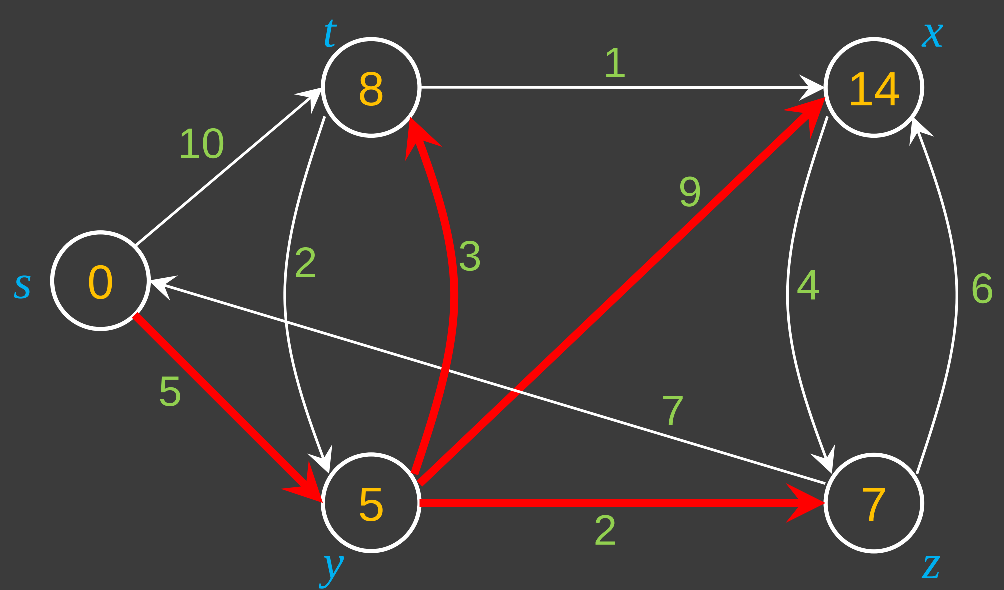
Relaxing (y, t) gives us a new $t.d$ of 8 and a new $t.\pi$ of y

Relaxing (y, z) gives us a new $z.d$ of 7 and a new $z.\pi$ of y

Relaxing (y, x) gives us a new $x.d$ of 14 and a new $x.\pi$ of y

We extract the vertex v from the front of Q, and add it to S. Then we relax all of the edges leading from v, $\{(y, t), (y, z), (y, x)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough

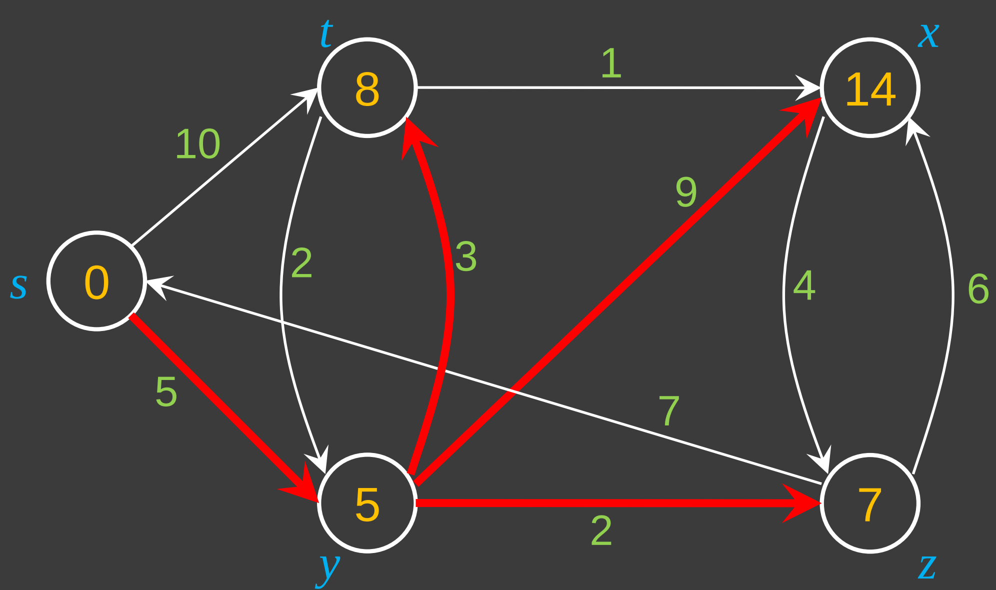


$Q = \{z(7), t(8), x(14)\}$
$S = \{s, y\}$

Relaxing (z, s) results in no change

We extract the vertex z from the front of Q, and add it to S. Then we relax all of the edges leading from z, $\{(z, s), (z, x)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough



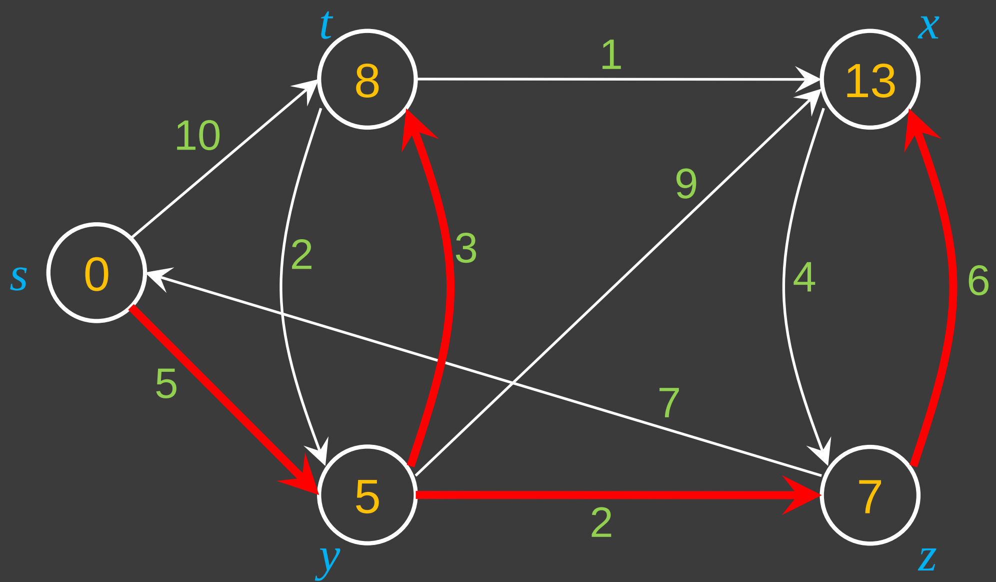
$Q = \{t(8), x(14)\}$
$S = \{s, y, z\}$

Relaxing (z, s) results in no change

Relaxing (z, x) gives us a new $x.d$ of 13 and a new $x.\pi$ of z

We extract the vertex z from the front of Q, and add it to S. Then we relax all of the edges leading from z, $\{(z, s), (z, x)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough

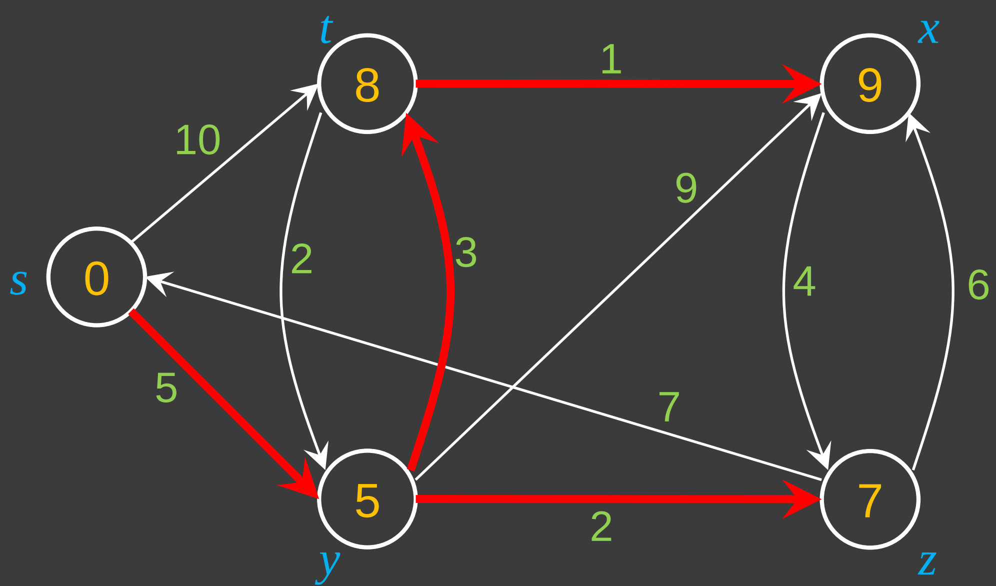


$Q = \{t(8), x(13)\}$
$S = \{s, y, z\}$

Relaxing (t, x) gives us a new $x.d$ of 9 and a new $x.\pi$ of t

We extract the vertex t from the front of Q, and add it to S. Then we relax all of the edges leading from t, $\{(t, x), (t, y)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough

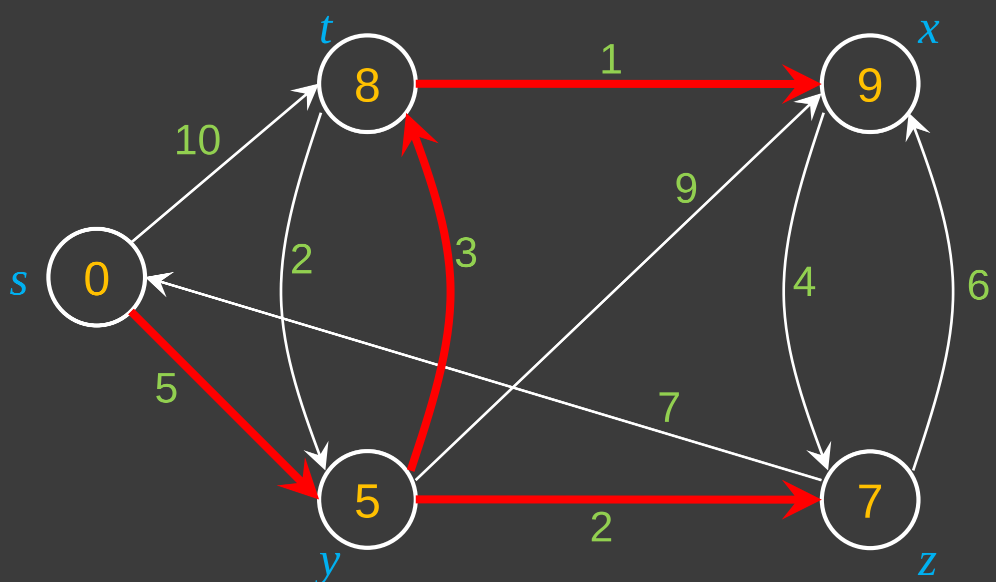


$Q = \{x(9)\}$
$S = \{s, y, z, t\}$

Relaxing (t, y) results in no change

We extract the vertex x from the front of Q, and add it to S. Then we relax all of the edges leading from x, $\{(x, z)\}$ updating the values in the queue corresponding to the vertices at the other ends of those edges.

DIJKSTRA Walkthrough

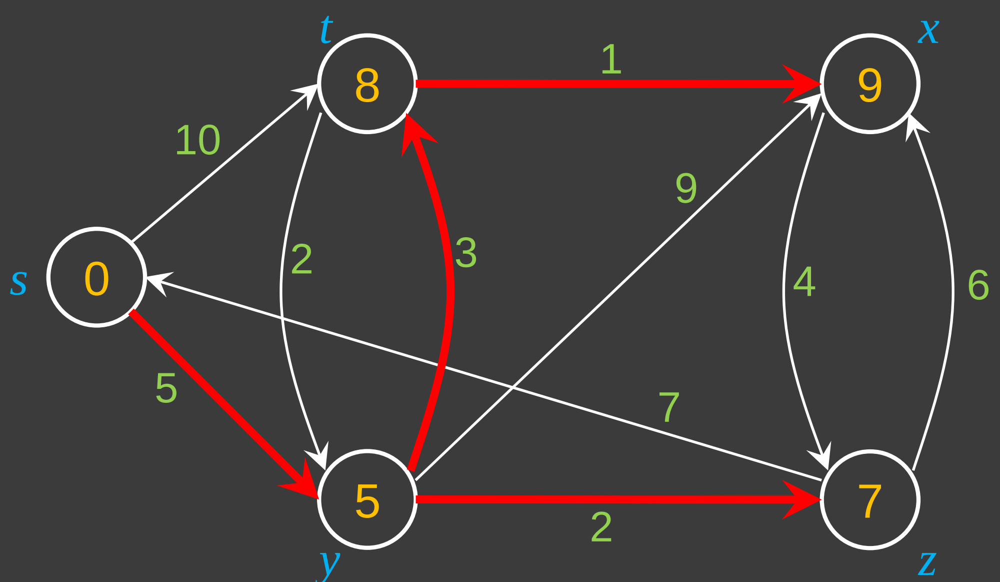


$Q = \{\}$
$S = \{s, y, z, t, x\}$

Relaxing (x, z) results in no change

Q is now empty, so the algorithm completes.

DIJKSTRA Walkthrough



So, the shortest-length path from s to…

…s is 0: $p = \langle \rangle$
…t is 8: $p = \langle s, y, t \rangle$
…x is 9: $p = \langle s, y, t, x \rangle$
…y is 5: $p = \langle s, y \rangle$
…z is 7: $p = \langle s, y, z \rangle$

DIJKSTRA's Algorithm – Comments

- DIJKSTRA's algorithm can be viewed as greedy, since it always chooses the "lightest", or "closest" vertex in $\{V - S\}$ to add to S (line 5)
- Each time through the **while** loop (lines 4-8), a vertex u is extracted from the priority queue
  - This vertex has the minimum distance from the source
  - The edges leaving u are then relaxed, which updates the distances in the priority queue

DIJKSTRA's Algorithm – Run Time

- The running time depends on implementation of priority queue.
  - If binary heap, each operation takes $O(\lg V)$ time, and there are E of them, so $O(E \lg V)$
  - If a Fibonacci heap:
    - Each EXTRACT-MIN takes $O(1)$ amortized time.
    - There are $O(V)$ other operations, taking $O(\lg V)$ amortized time each.
    - Therefore, time is $O(V \lg V + E)$