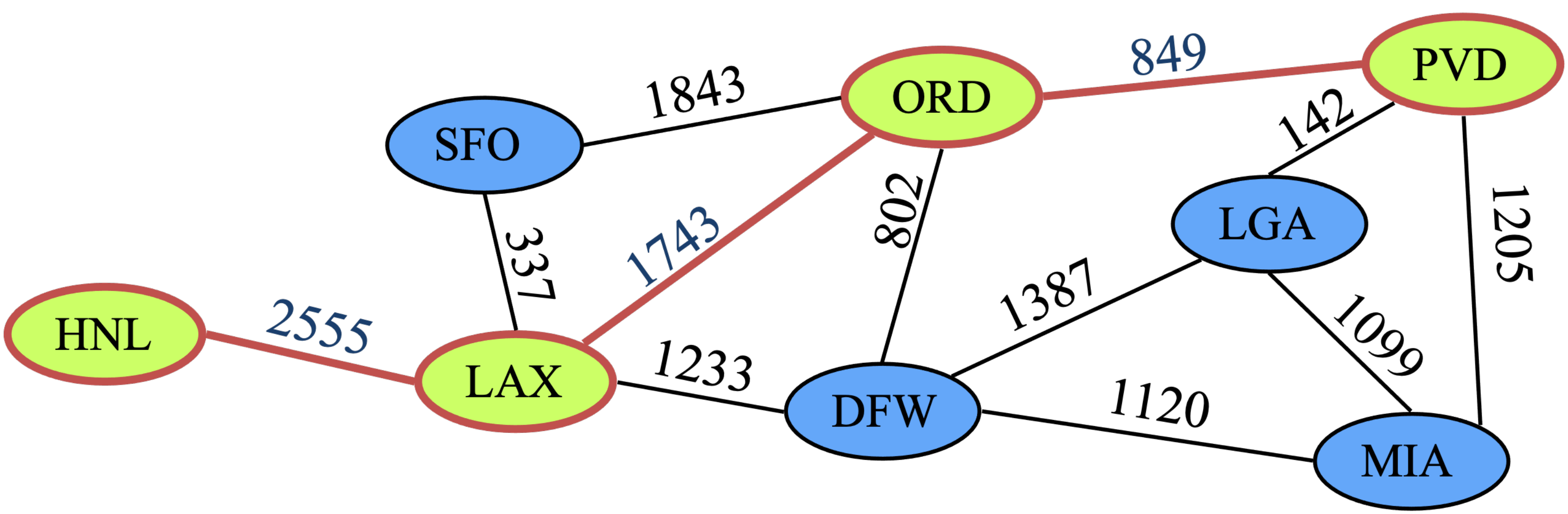
Single-Source Shortest Path

CLRS 22

(+ some supplemental material)

Graph

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v***.
  - Length of a path is the sum of the weights of its edges
- Example: shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Shortest Paths

How to find the shortest route between two points on a map.

Input:

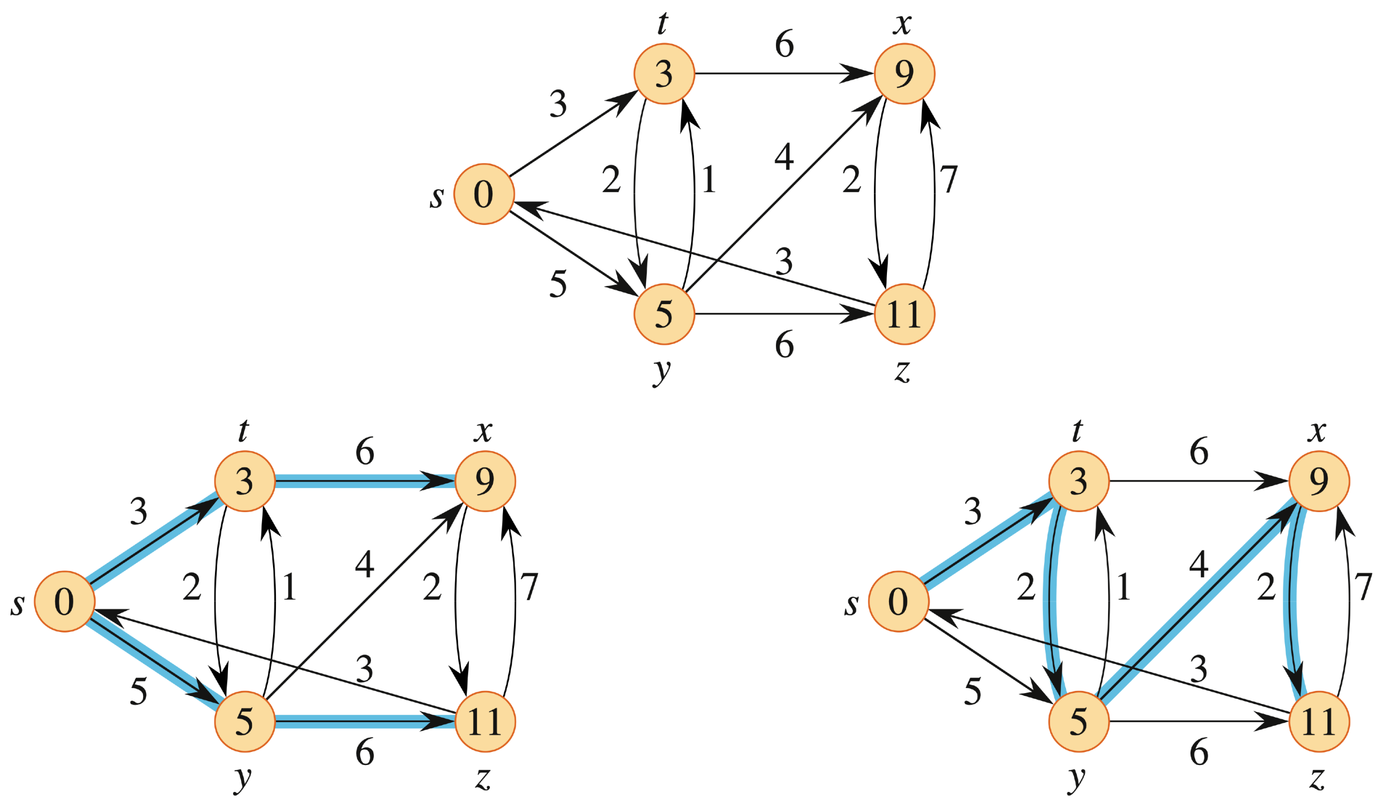
- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Shortest-path weight u to v:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example: shortest paths from s

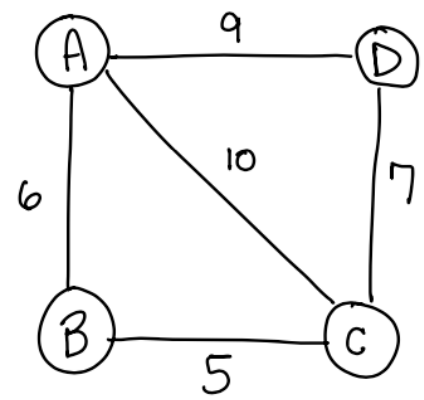


This example shows that a shortest path might not be unique.

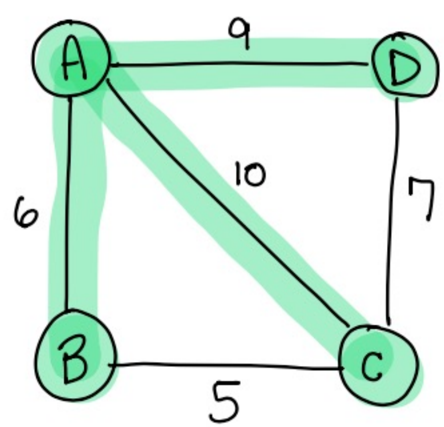
It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Shortest Path Trees != Minimum Spanning Trees

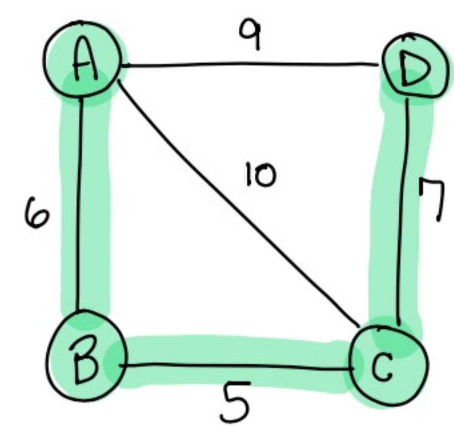
Consider the following graph.



Shortest path tree (rooted at A)



MST



Negative Weight Edges

OK, as long as no negative-weight cycles are reachable from the source.

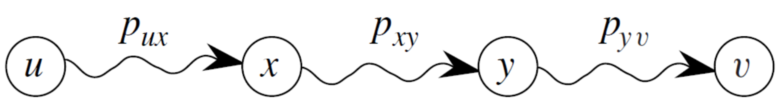
- If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph. We'll be clear when they're allowed and not allowed.

OPTIMAL SUBSTRUCTURE

Lemma

Any subpath of a shortest path is a shortest path.

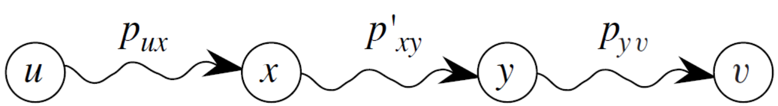
Proof Cut-and-paste.



Now suppose there exists a shorter path $x \overset{p'_{xy}}{\leadsto} y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p':



Contradicts the assumption that p is a shortest path.

CYCLES

Shortest paths can’t contain cycles:

- Already ruled out negative-weight cycles.
- Positive-weight $\Rightarrow$ we can get a shorter path by omitting the cycle.
- 0-weight: no reason to use them $\Rightarrow$ assume that our solutions won’t use them.

OUTPUT OF SINGLE-SOURCE SHORTEST-PATH ALGORITHM

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi = \text{NIL}$.
  - π induces a tree—***shortest-path tree***.

INITIALIZATION

All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

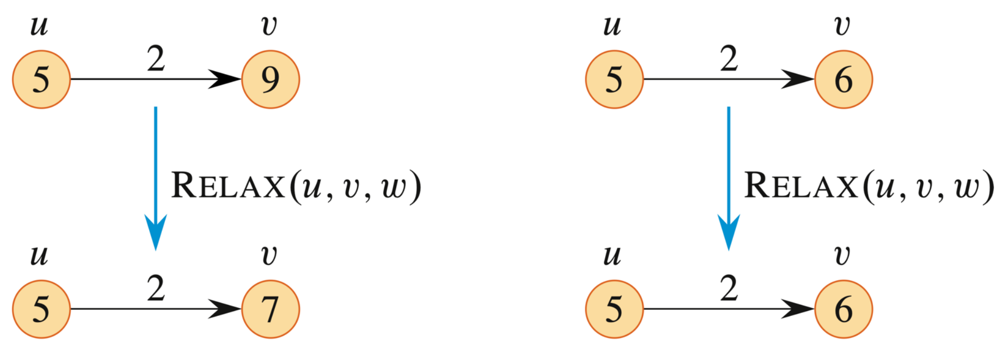
INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

RELAXING AN EDGE (u,v)

Can the shortest-path estimate for v be improved by going through u and taking (u, v)?

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



(a) (b)

RELAXING AN EDGE (continued)

For all the single-source shortest-paths algorithms we'll look at,

- start by calling INITIALIZE-SINGLE-SOURCE,
- then relax edges.

The algorithms differ in the order and how many times they relax each edge.

SHORTEST-PATHS PROPERTIES

Based on calling INITIALIZE-SINGLE-SOURCE once and then calling RELAX zero or more times.

Triangle inequality: For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property: Always have $v.d \geq \delta(s, v)$ for all v. Once $v.d$ gets down to $\delta(s, v)$, it never changes.

No-path property: If $\delta(s, v) = \infty$, then $v.d = \infty$ always.

Convergence property: If $s \leadsto u \to v$ is a shortest path, $u.d = \delta(s, u)$, and edge (u, v) is relaxed, then $v.d = \delta(s, v)$ afterward.

Path-relaxation property: Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

THE BELLMAN-FORD ALGORITHM

- Allows negative-weight edges.
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

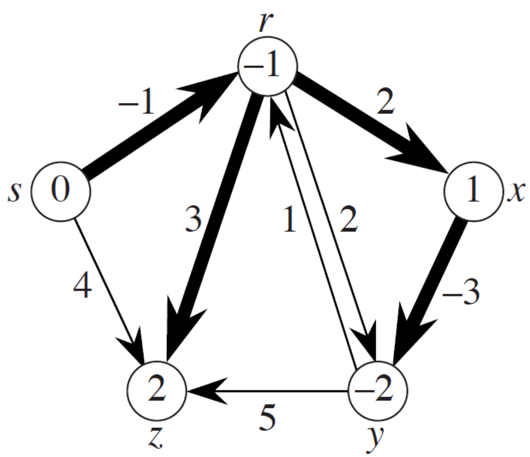
THE BELLMAN-FORD ALGORITHM
(continued)

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Time: $O(V^2 + VE)$. The first **for** loop makes $|V| - 1$ passes over the edges, and each pass takes $\Theta(V + E)$ time. We use O rather than Θ because sometimes $< |V| - 1$ passes are enough (Exercise 22.1-3).

So, in a connected graph Bellman-Ford runs in **O(nm) time**

EXAMPLE

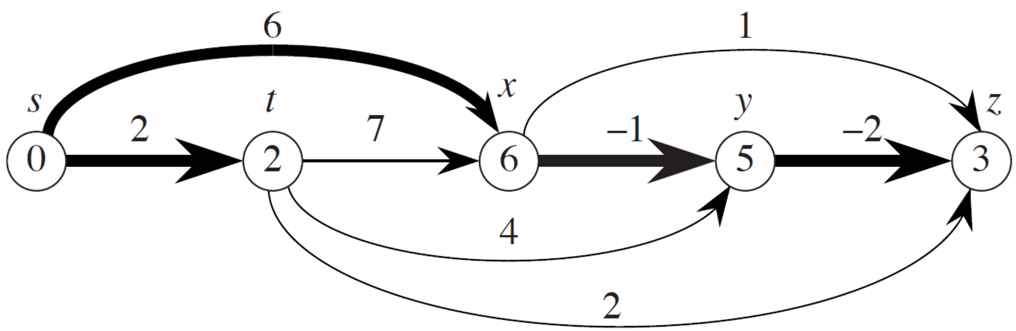


SINGLE-SOURCE SHORTEST PATHS IN A DIRECTED ACYCLIC GRAPH

Since a dag, we're guaranteed no negative-weight cycles.

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u ∈ G.V, taken in topologically sorted order
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
```

EXAMPLE



Time

$\Theta(V + E)$.

Correctness

Because vertices are processed in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.
$\Rightarrow$ Edges on any shortest path are relaxed in order.
$\Rightarrow$ By path-relaxation property, correct.

So, in a connected DAG, the DAG-based algorithm runs in **O(m) time**

DIJKSTRA'S ALGORITHM

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($v.d$).
- Can think of waves, like BFS.
- A wave emanates from the source.
- The first time that a wave arrives at a vertex, a new wave emanates from that vertex.

Have two sets of vertices:

- $S =$ vertices whose final shortest-path weights are determined,
- $Q =$ priority queue $= V - S$.

DIJKSTRA'S ALGORITHM (continued)

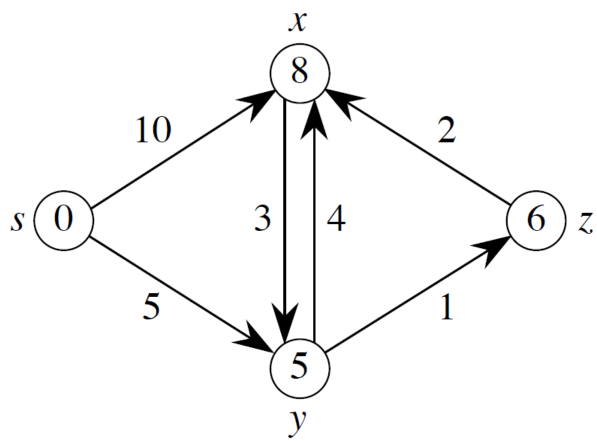
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

DIJKSTRA'S ALGORITHM (continued)

- Looks a lot like Prim's algorithm, but computing $v.d$, and using shortest-path weights as keys.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest"?) vertex in $V - S$ to add to S.

Like Prim's algorithm, Dijkstra's algorithm runs in **O(m log n) time** on a connected graph if we use a binary heap to implement the priority queue.

EXAMPLE



Order of adding to S: s, y, z, x.

Correctness

The algorithm extracts vertices from the heap in order of shortest distance from the source. Inductively, if the algorithm has found the shortest paths to some set *S*, the shortest path to the closest vertex in *V-S* can be found by appending a single edge to a path to some vertex in *S*.