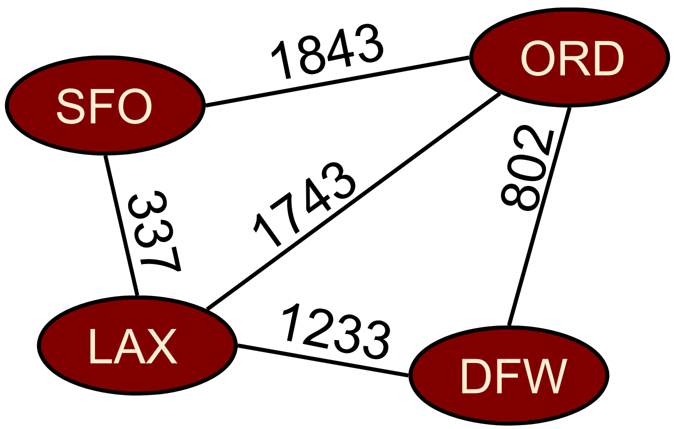
Graphs – Shortest Path (Weighted Graph)



Outline

- The shortest path problem
- Single-source shortest path
  - Shortest path on a directed acyclic graph (DAG)
  - Shortest path on a general graph: Dijkstra's algorithm

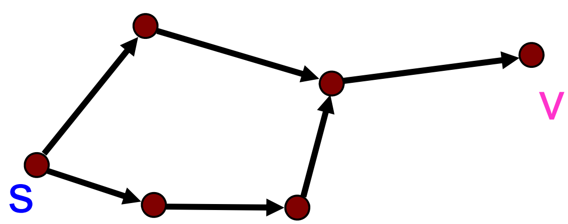
Outline

- **The shortest path problem**
- Single-source shortest path
  - Shortest path on a directed acyclic graph (DAG)
  - Shortest path on a general graph: Dijkstra’s algorithm

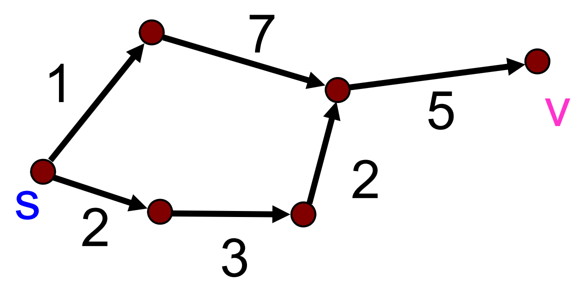
Shortest Path on Weighted Graphs

- BFS finds the **shortest paths** from a source node **s** to every vertex **v** in the graph.
- Here, the **length** of a path is simply the number of edges on the path.
- But what if edges have different ‘costs’?

$\delta(s,v) = 3$

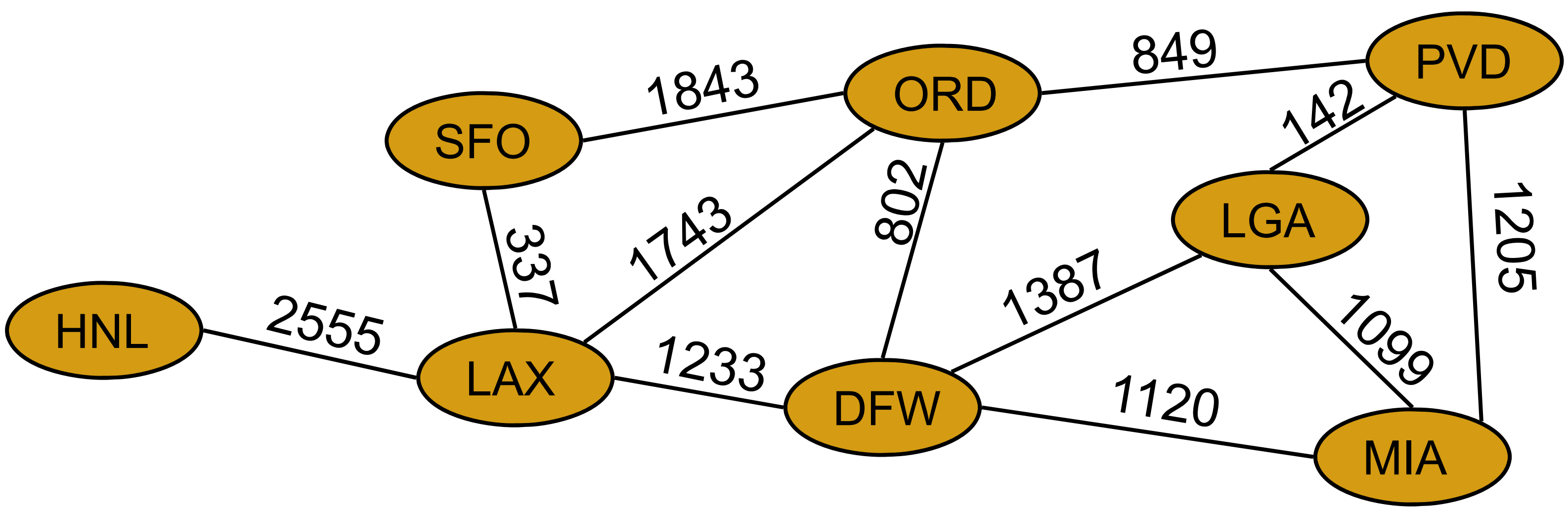


$\delta(s,v) = 12$



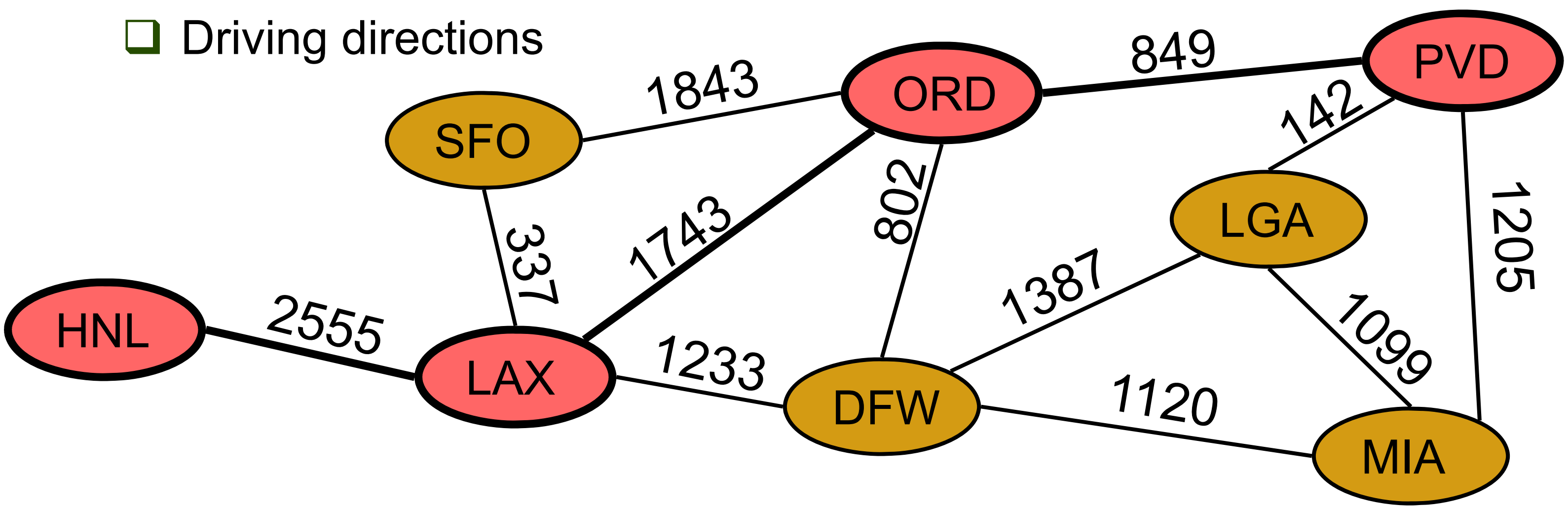
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Path on a Weighted 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 Path: Notation

- Input:

Directed Graph $G = (V, E)$

Edge weights $w : E \rightarrow \mathbf{N}$

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle \quad = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

Shortest-path weight from u to v :

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

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

Shortest Path Properties

Property 1 (Optimal Substructure):

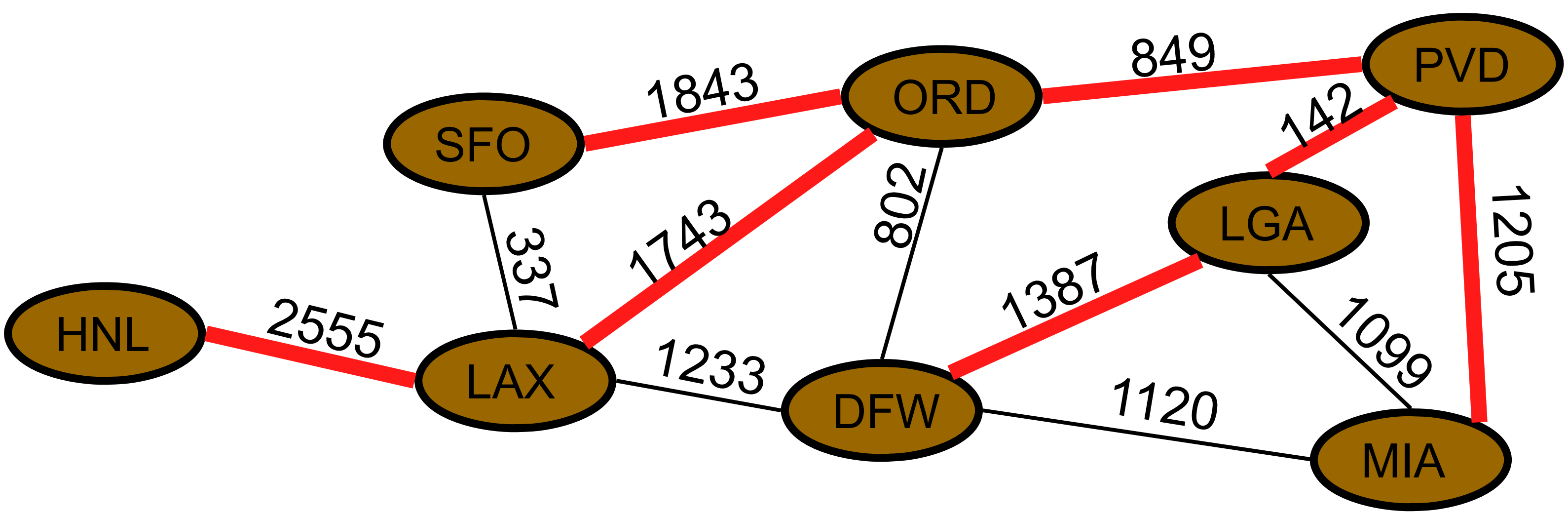
A subpath of a shortest path is itself a shortest path

Property 2 (Shortest Path Tree):

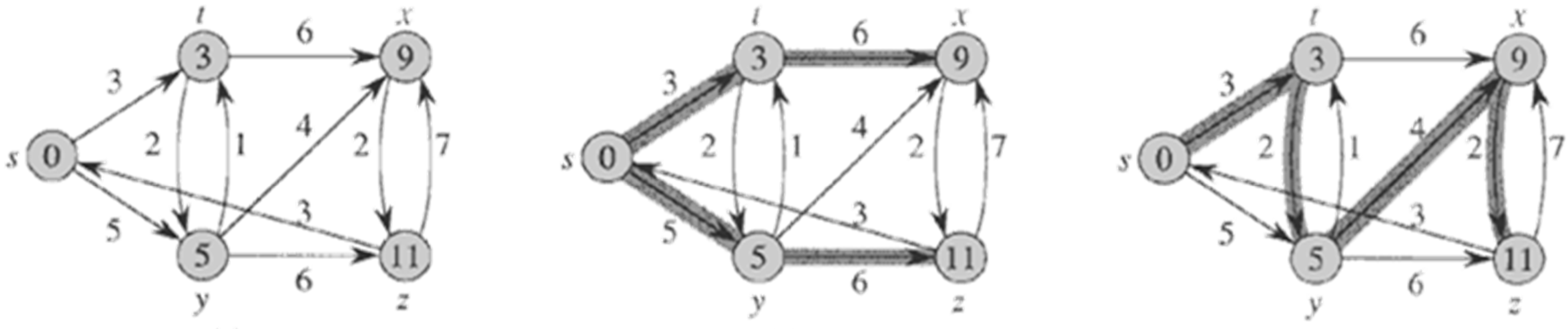
There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



Shortest path trees are not necessarily unique



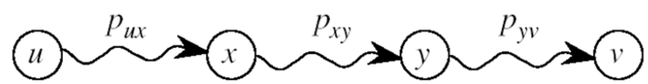
(a) (b) (c)

Single-source shortest path search induces a search tree rooted at *s*.

This tree, and hence the paths themselves, are not necessarily unique.

Optimal substructure: Proof

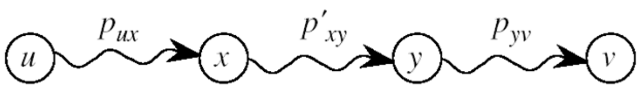
- Lemma: Any subpath of a shortest path is a shortest path
- Proof: Cut and paste.

Suppose this path p is a shortest path from u to v. 

Then $\delta(u,v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

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

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

Construct p′: 

Then $w(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \;<\; w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \;= w(p)$.

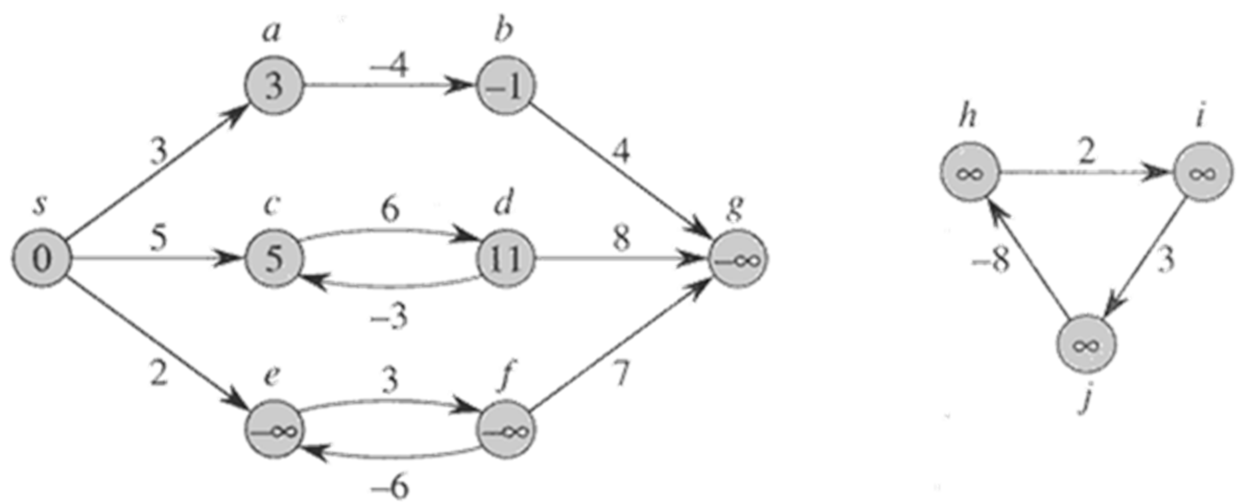
So p wasn't a shortest path after all!

Shortest path variants

- **Single-source shortest-paths problem:** – the shortest path from *s* to each vertex *v*.
- **Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex *t* from each vertex *v*.
- **Single-pair shortest-path problem:** Find a shortest path from *u* to *v* for given vertices *u* and *v*.
- **All-pairs shortest-paths problem:** Find a shortest path from *u* to *v* for every pair of vertices *u* and *v*.

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) = −∞ 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.



Cycles

- Shortest paths can’t contain cycles:
  - Already ruled out negative-weight cycles.
  - Positive-weight: we can get a shorter path by omitting the cycle.
  - Zero-weight: no reason to use them → assume that our solutions won’t use them.

Outline

- The shortest path problem
- **Single-source shortest path**
  - Shortest path on a directed acyclic graph (DAG)
  - Shortest path on a general graph: Dijkstra's algorithm

Output of a single-source shortest-path algorithm

- For each vertex v in V:
  - $d[v] = \delta(s, v)$.
    - Initially, $d[v]=\infty$.
    - Reduce as algorithm progresses. But always maintain $d[v] \geq \delta(s, v)$.
    - Call $d[v]$ a shortest-path estimate.
  - $\pi[v]$ = predecessor of v on a shortest path from s.
    - If no predecessor, $\pi[v]$ = NIL.
    - π induces a tree — **shortest-path tree**.

Initialization

- All shortest-path algorithms start with the same initialization:

INIT-SINGLE-SOURCE(V, s)

for each v in V

    do d[v] $\leftarrow \infty$

        π[v] $\leftarrow$ NIL

d[s] $\leftarrow$ 0

Relaxing an edge

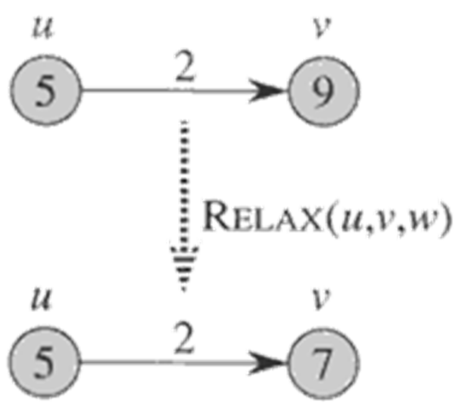
- Can we improve shortest-path estimate for v by first going to ***u*** and then following edge ***(u,v)***?

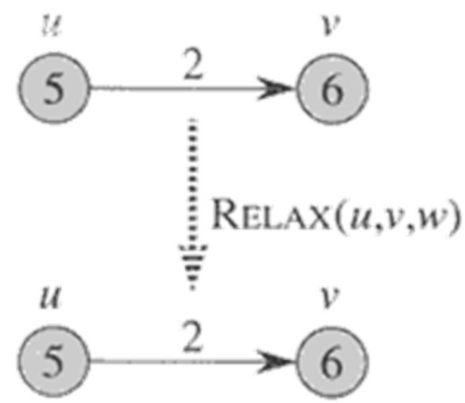
RELAX(u, v, w)

    if d[v] > d[u] + w(u, v) then

        d[v] ← d[u] + w(u, v)

        π[v]← u





General single-source shortest-path strategy

1. Start by calling INIT-SINGLE-SOURCE
2. Relax Edges

Algorithms differ in the order in which edges are taken and how many times each edge is relaxed.

Outline

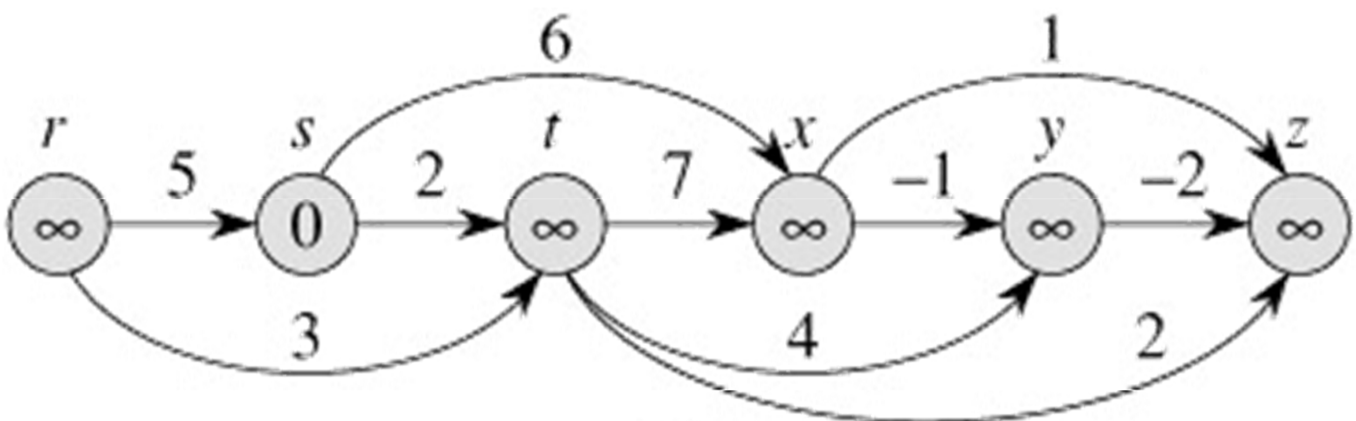
- The shortest path problem
- Single-source shortest path
  - **Shortest path on a directed acyclic graph (DAG)**
  - Shortest path on a general graph: Dijkstra's algorithm

Example 1. Single-Source Shortest Path on a Directed Acyclic Graph

- Basic Idea: topologically sort nodes and relax in linear order.
- Efficient, since *δ[u]* (shortest distance to *u*) has already been computed when edge (u,v) is relaxed.
- Thus we only relax each edge once, and never have to backtrack.

Example: Single-source shortest paths in a directed acyclic graph (DAG)

- Since graph is a DAG, we are guaranteed no negative-weight cycles.
- Thus algorithm can handle negative edges



(a)

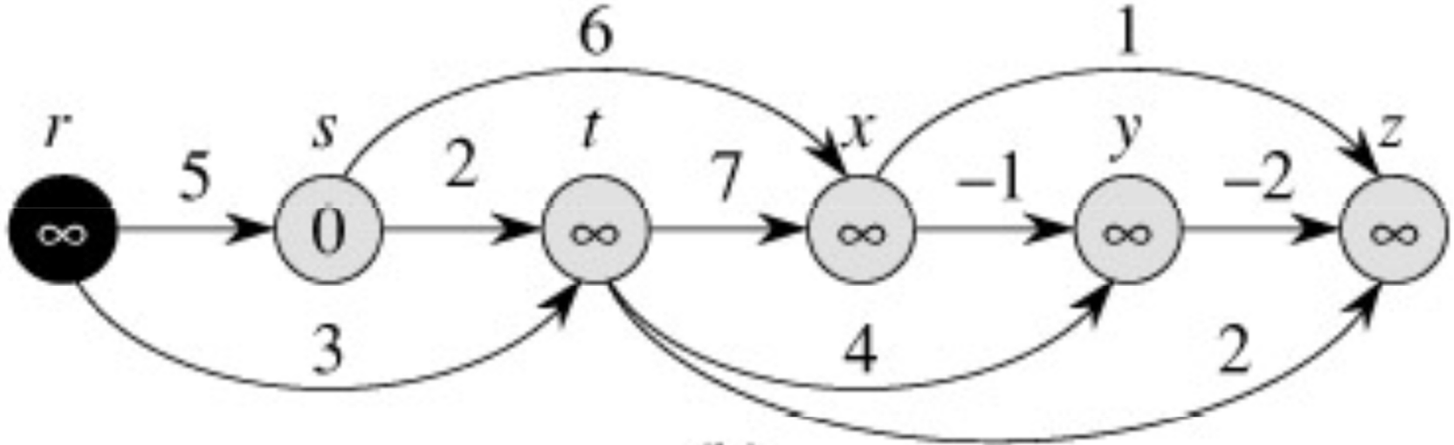
Algorithm

DAG-SHORTEST-PATHS(G, w, s)

topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    do for each vertex $v \in Adj[u]$
        do RELAX(u, v, w)

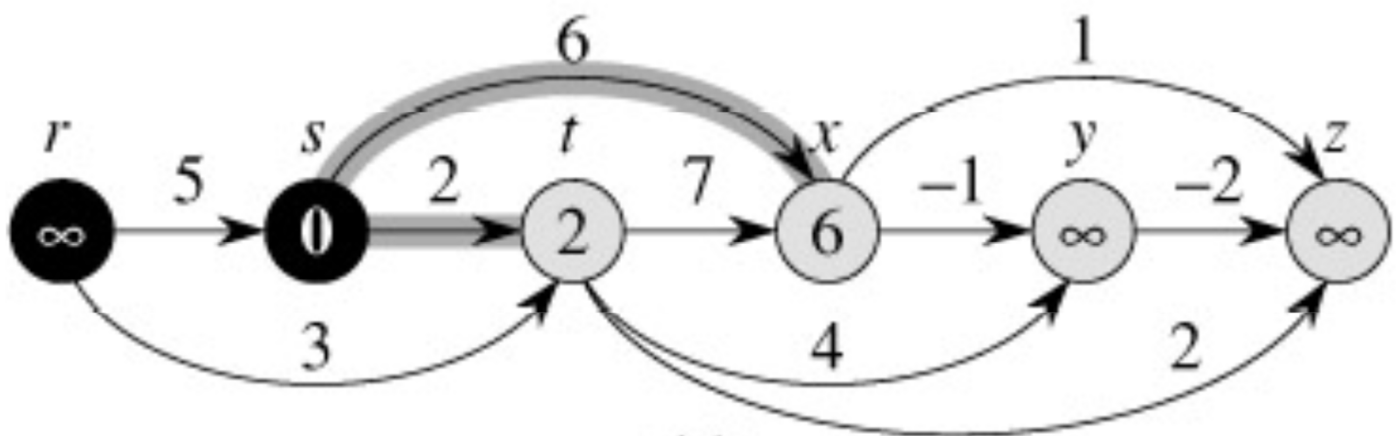
Time: $\Theta(V + E)$

Example



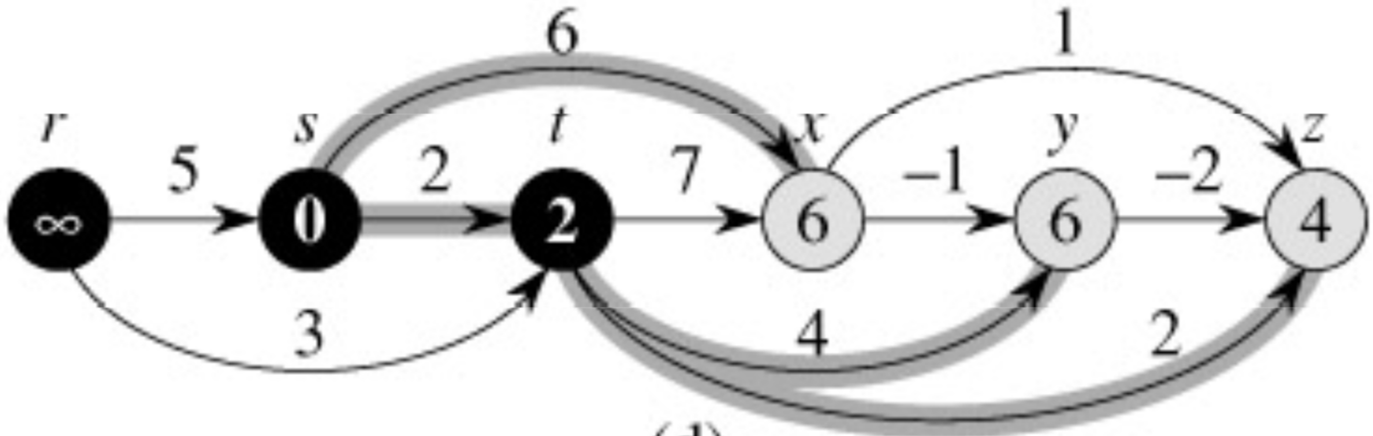
(b)

Example



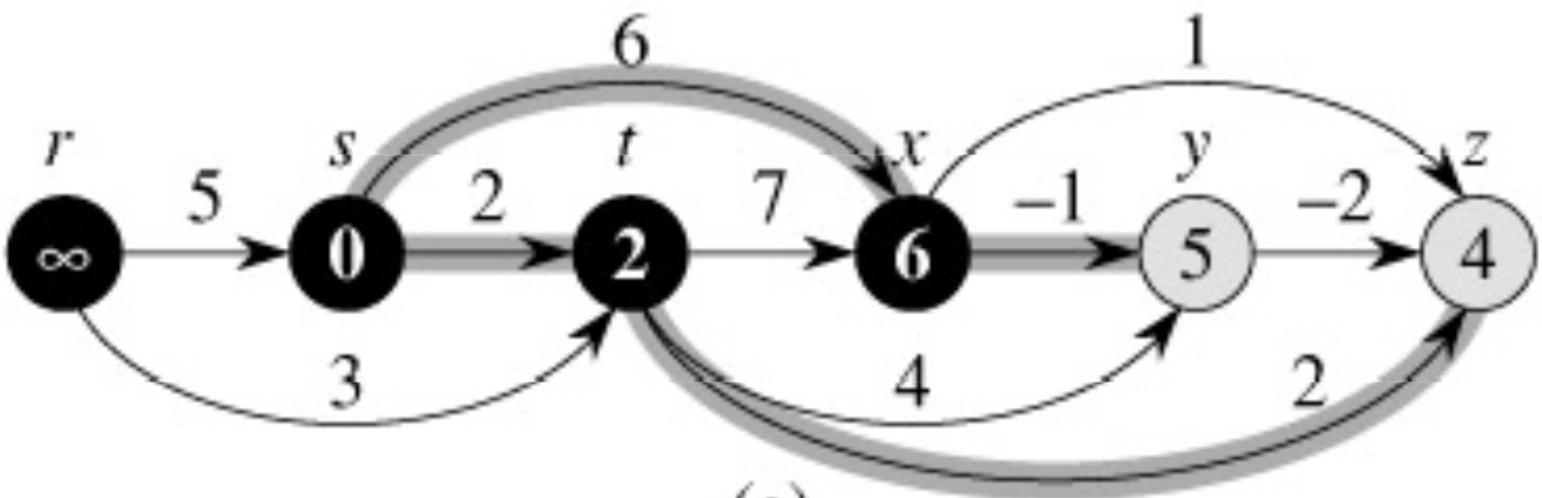
(c)

Example



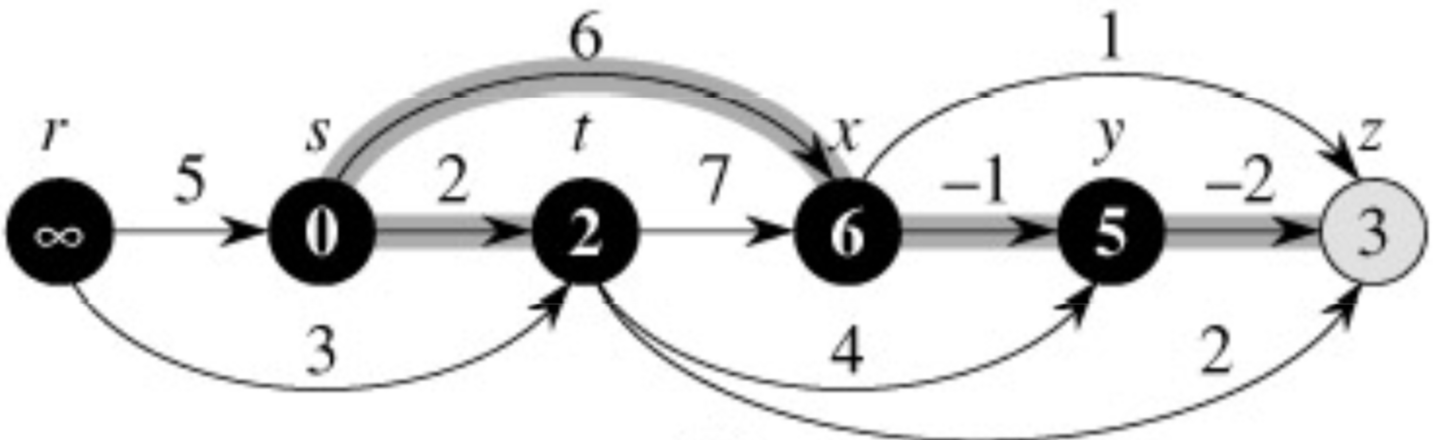
(d)

Example



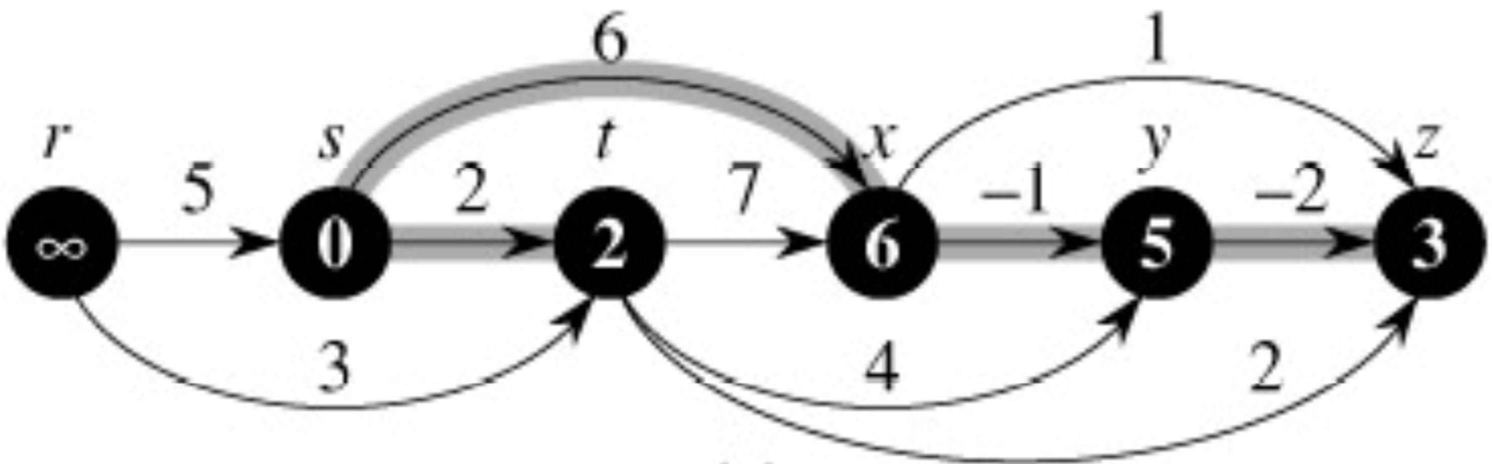
(e)

Example



(f)

Example



(g)

Correctness: 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 we relax, in order, (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k),

even intermixed with other relaxations,

then $d[v_k] = \delta(s,\ v_k)$.

Correctness of DAG Shortest Path Algorithm

- Because we process vertices in topologically sorted order, edges of *any* path are relaxed in order of appearance in the path.
  - →Edges on any shortest path are relaxed in order.
  - →By path-relaxation property, correct.

Outline

- The shortest path problem
- Single-source shortest path
  - Shortest path on a directed acyclic graph (DAG)
  - **Shortest path on a general graph: Dijkstra's algorithm**

Example 2. Single-Source Shortest Path on a General Graph (May Contain Cycles)

- This is fundamentally harder, because the first paths we discover may not be the shortest (not monotonic).

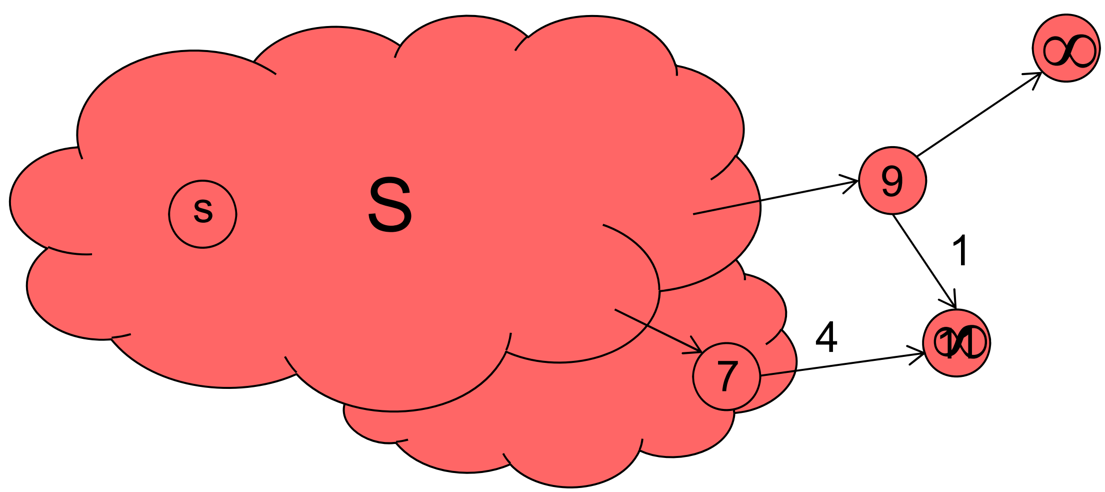
Dijkstra's algorithm (E. Dijkstra,1959)

- Applies to general, weighted, directed or undirected graph (may contain cycles).
- But weights must be non-negative. (But they can be 0!)
- Essentially a weighted version of BFS.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights (d[v]).
- Maintain 2 sets of vertices:
  - S = vertices whose final shortest-path weights are determined.
  - Q = priority queue = V-S.

Edsger Dijkstra

Dijkstra's Algorithm: Operation

- We grow a "**cloud**" S of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud S and its adjacent vertices
- At each step
  - We add to the cloud S the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u



Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S ← Ø
  Q ← V[G]
  while Q ≠ Ø
        do u ← EXTRACT-MIN(Q)
           S ← S ∪ {u}
           for each vertex v ∈ Adj[u]
               do RELAX(u, v, w)
```

- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" vertex in V – S to add to S.

Dijkstra's algorithm: Analysis

- Analysis:
  - Using minheap, queue operations takes *O(logV)* time

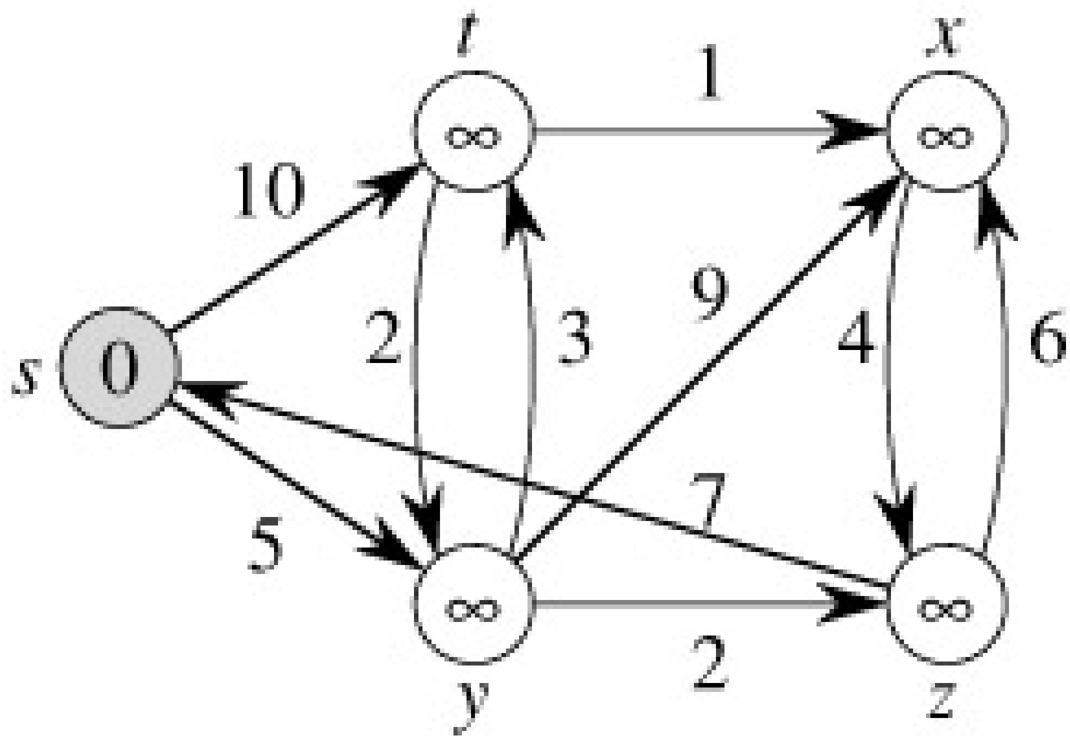
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)                O(V)
S ← Ø
Q ← V[G]
while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)                   O(logV)×O(V) iterations
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             do RELAX(u, v, w)                O(logV)×O(E) iterations
```

→ Running Time is $O(E \log V)$

Example

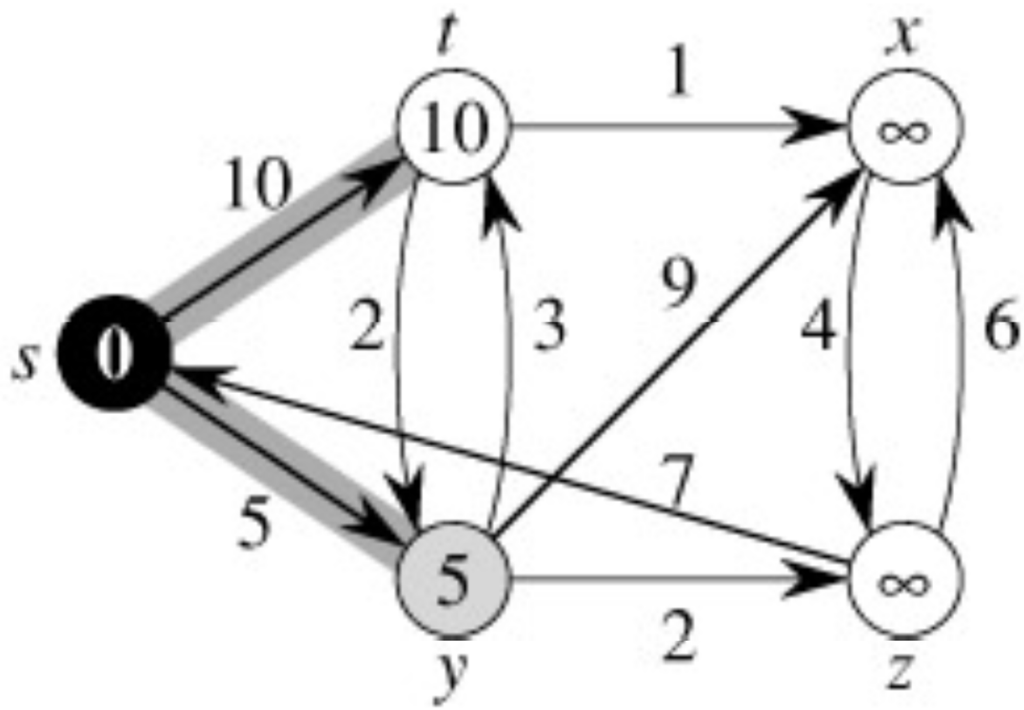
Key:

White	$\Leftrightarrow$	Vertex $\in Q = V - S$
Grey	$\Leftrightarrow$	Vertex = min(Q)
Black	$\Leftrightarrow$	Vertex $\in S$, Off Queue



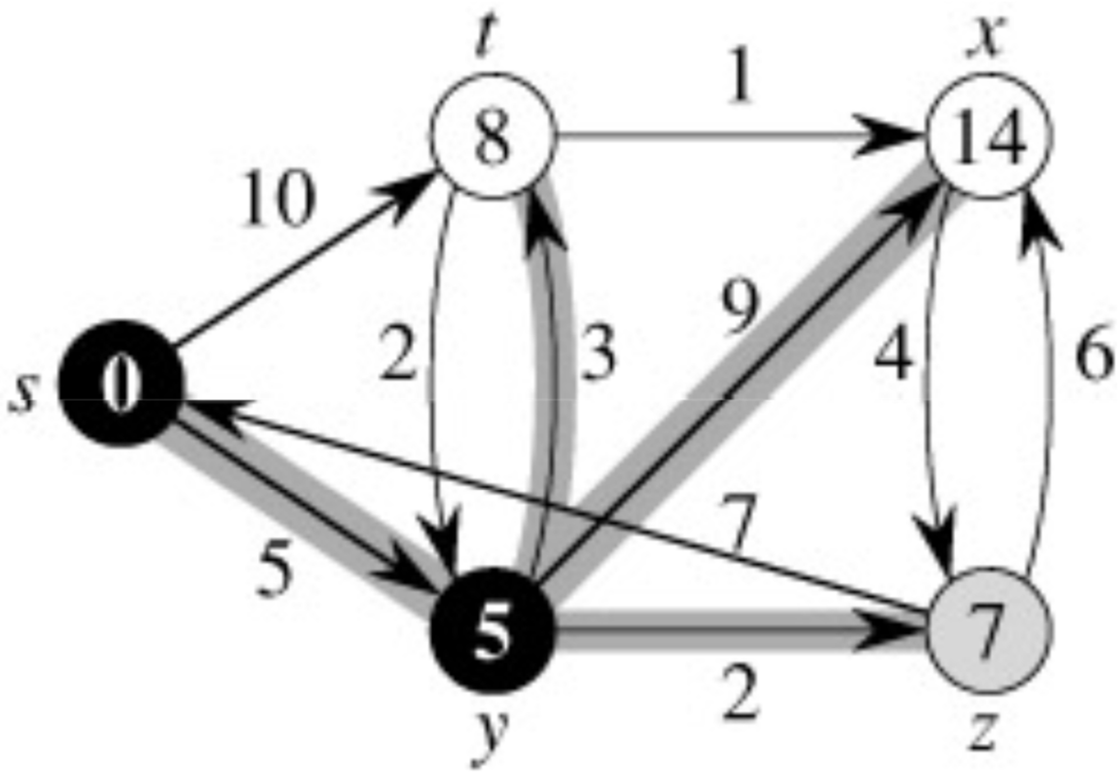
(a)

Example



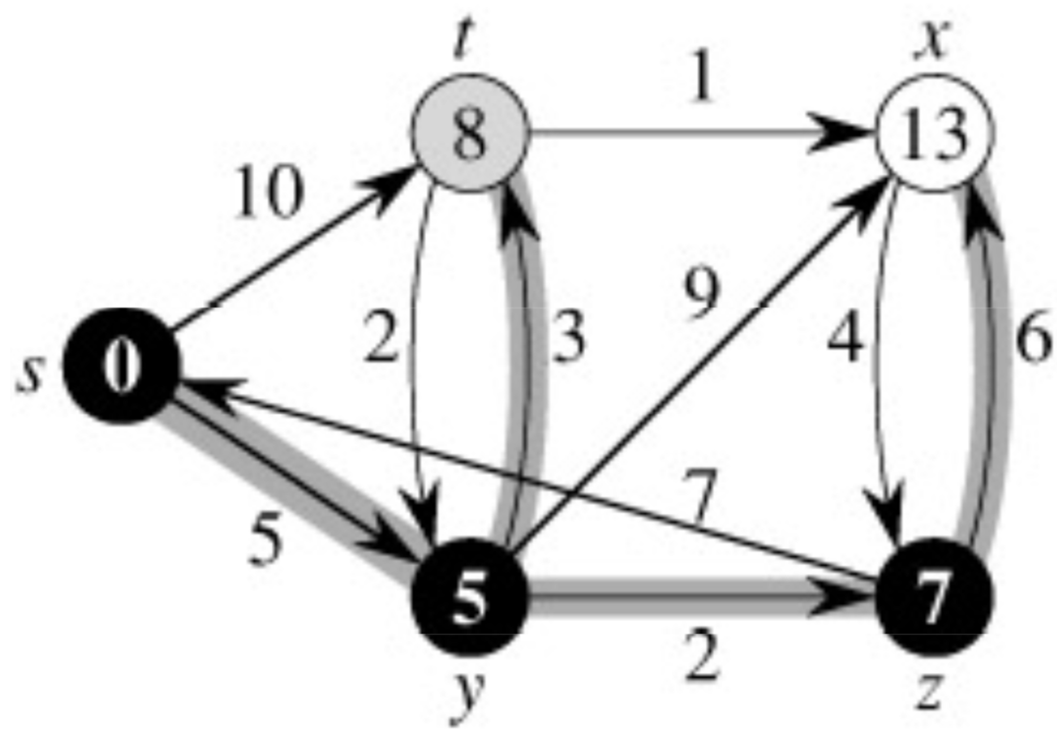
(b)

Example



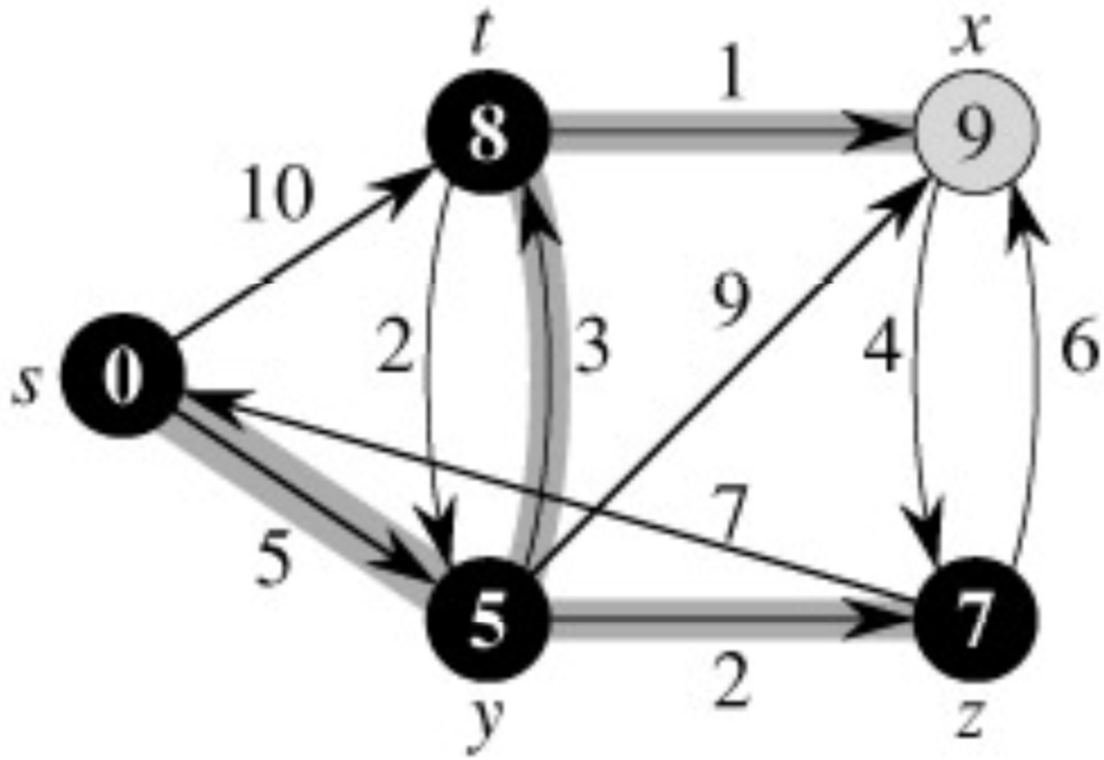
(c)

Example



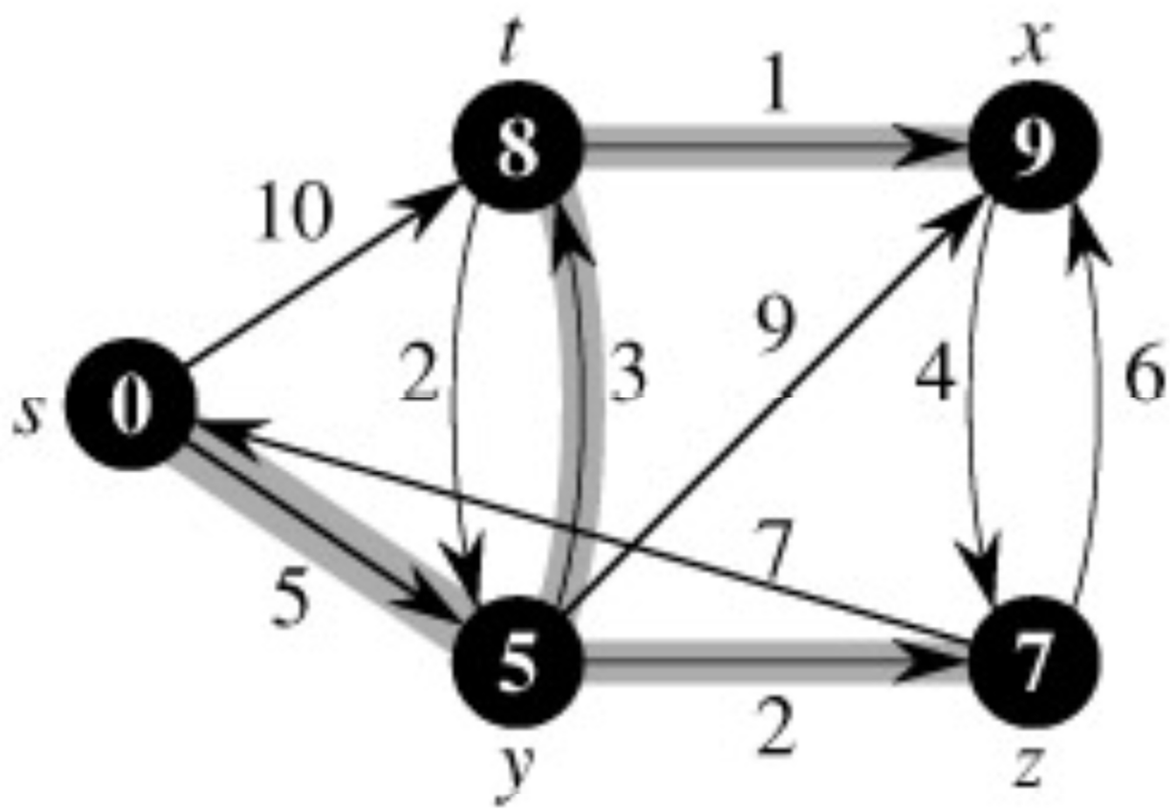
(d)

Example



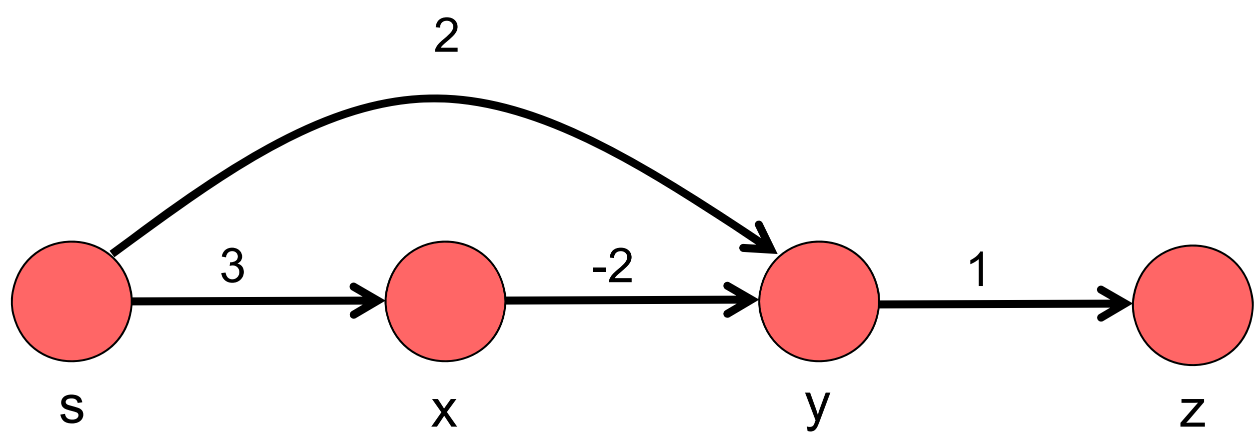
(e)

Example



(f)

Djikstra's Algorithm Cannot Handle Negative Edges



Correctness of Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
        S ← S ∪ {u}
        for each vertex v ∈ Adj[u]
            do RELAX(u, v, w)
```

- **Loop invariant:** d[v] = δ(s, v) for all v in S.
  - Initialization: Initially, S is empty, so trivially true.
  - Termination: At end, Q is empty → S = V → d[v] = δ(s, v) for all v in V.
  - Maintenance:
    - Need to show that
      - d[u] = δ(s, u) when u is added to S in each iteration.
      - d[u] does not change once u is added to S.

Correctness of Dijkstra's Algorithm: Upper Bound Property

- Upper Bound Property:

  1. $d[v] \geq \delta(s,v) \forall v \in V$
  2. Once $d[v] = \delta(s,v)$, it doesn't change

- Proof:

  By induction.

  Base Case: $d[v] \geq \delta(s,v) \forall v \in V$ immediately after initialization, since

  $d[s] = 0 = \delta(s,s)$

  $d[v] = \infty \forall v \neq s$

  Inductive Step:

  Suppose $d[x] \geq \delta(s,x) \forall x \in V$

  Suppose we relax edge (u,v).

  If $d[v]$ changes, then $d[v] = d[u] + w(u,v)$

  $$\geq \delta(s,u) + w(u,v)$$

  A valid path from s to v!

  $$\geq \delta(s,v)$$

Correctness of Dijkstra's Algorithm

Claim: When u is added to S, $d[u] = \delta(s,u)$

Proof by Contradiction: Let u be the first vertex added to S such that $d[u] \neq \delta(s,u)$ when u is added.

Let y be first vertex in $V - S$ on shortest path to u

Let x be the predecessor of y on the shortest path to u

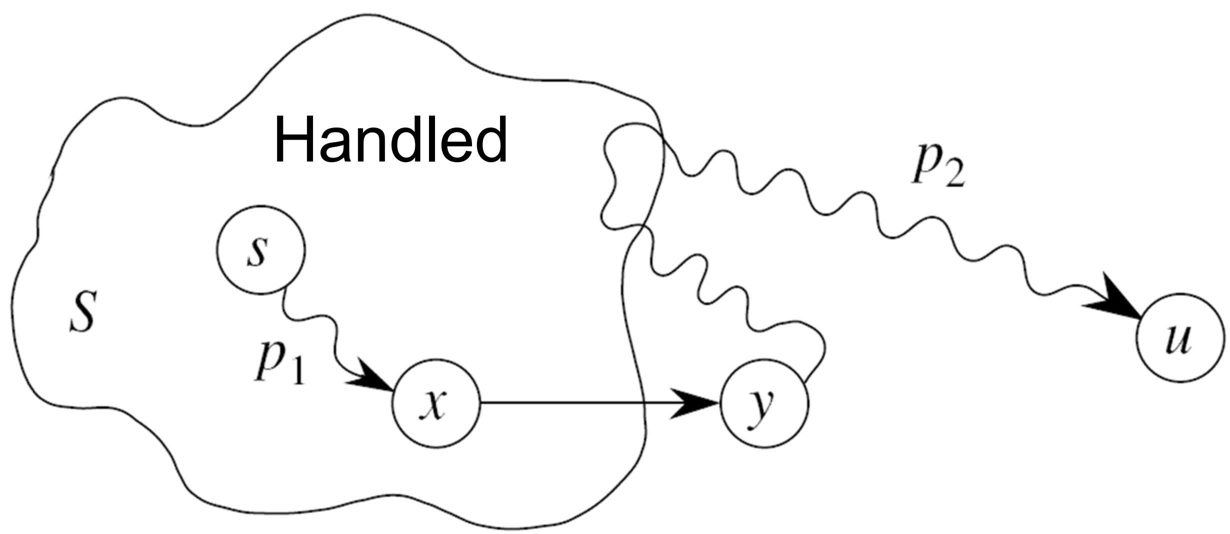
Claim: $d[y] = \delta(s,y)$ when u is added to S.

Proof:

$d[x] = \delta(s,x)$, since $x \in S$.

(x,y) was relaxed when x was added to S $\rightarrow d[y] = \delta(s,x) + w(x,y) = \delta(s,y)$

Optimal substructure property!



Correctness of Dijkstra's Algorithm

Thus $d[y] = \delta(s, y)$ when u is added to S.

$\rightarrow d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$ (upper bound property)

But $d[u] \leq d[y]$ when u added to S

Thus $d[y] = \delta(s, y) = \delta(s, u) = d[u]$!

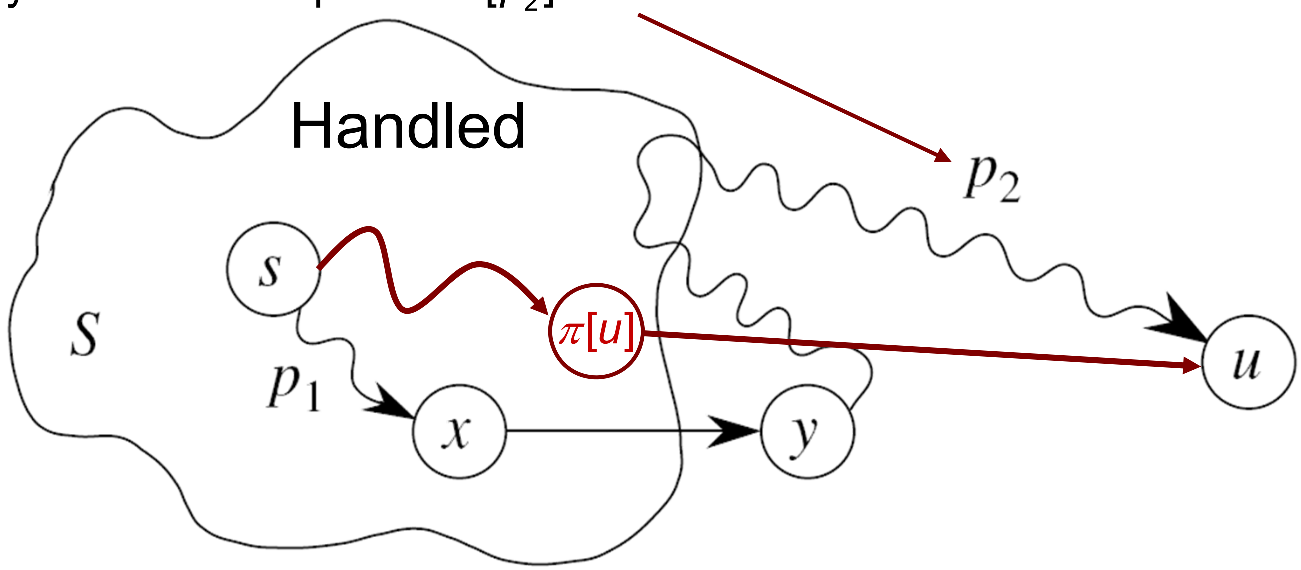
Thus when u is added to S, $d[u] = \delta(s, u)$

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← Ø
Q ← V[G]
while Q ≠ Ø
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             do RELAX(u, v, w)
```

Consequences:

There is a shortest path to u such that the predecessor of u $\pi[u] \in S$ when u is added to S.

The path through y can only be a shortest path if $w[p_2] = 0$.



Correctness of Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S ← ∅
  Q ← V[G]
  while Q ≠ ∅
      do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             do RELAX(u, v, w)
```

Relax(u,v,w) can only decrease $d[v]$.

By the upper bound property, $d[v] \geq \delta(s,v)$.

Thus once $d[v] = \delta(s,v)$, it will not be changed.

- **Loop invariant:** d[v] = δ(s, v) for all v in S.
  - Maintenance:
    - Need to show that
      - d[u] = δ(s, u) when u is added to S in each iteration. ✓
      - d[u] does not change once u is added to S. ?

Outline

- The shortest path problem
- Single-source shortest path
  - Shortest path on a directed acyclic graph (DAG)
  - Shortest path on a general graph: Dijkstra’s algorithm