CS 421: Design and Analysis of Algorithms

Chapter 24: Single-Source Shortest Paths

Dr. Gaby Dagher
Department of Computer Science
Boise State University

December 7, 2020

[Part of the lecture notes is based on materials by David Luebke]

Content of this Chapter

- Introduction
- The Bellman-Ford algorithm
- Single-source shortest paths in DAGs
- Dijkstra’s algorithm

Content of this Chapter

- **Introduction**
- The Bellman-Ford algorithm
- Single-source shortest paths in DAGs
- Dijkstra's algorithm

Introduction

Generalization of BFS to handle weighted graphs

- Directed Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$
- Edge weight function: $w : \boldsymbol{E} \rightarrow \boldsymbol{R}$
- In BFS $w(e)$=1 for all $e \in$ E

Weight of path p = $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{\mathrm{k}}$ is:

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

Shortest Path

- Shortest Path = Path of minimum weight

- Weight of a shortest path:

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

Shortest-Path Variants

- Shortest-Path problems
  - Single-source shortest-paths problem: Find the shortest path from *s* to each vertex *v*. (e.g. BFS)
  - 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*.

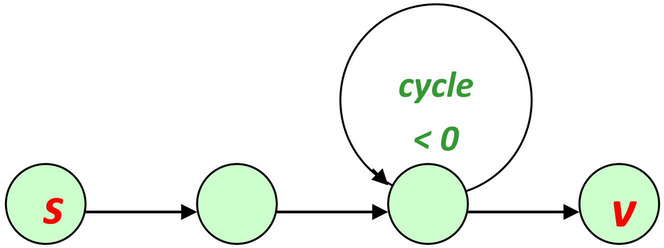
Weight of the Shortest Path

Definition:

- **$\delta(u,v)$ = weight of the shortest path(s) from *u* to *v***

Well Definedness:

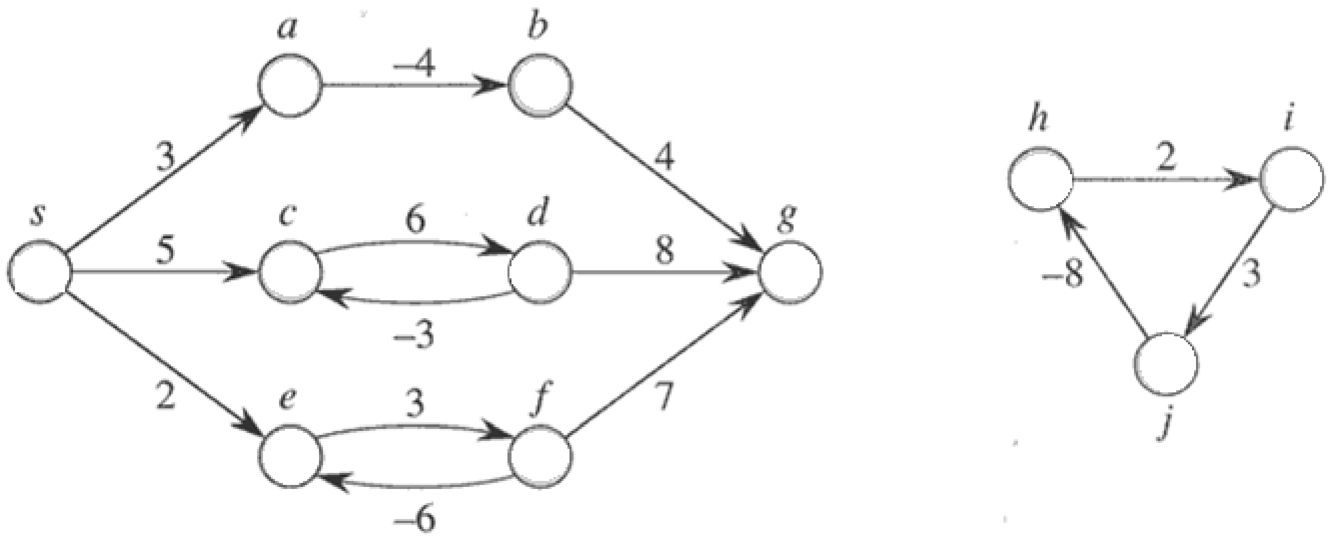
- *negative-weight cycle in graph*: Some shortest paths may *not be defined*.
- *Why?* We can always get a shorter path by going around the cycle again.



Negative-weight edges

- No problem, as long as no negative-weight cycles are reachable from the source.
- Otherwise, we can just keep going around it, and get

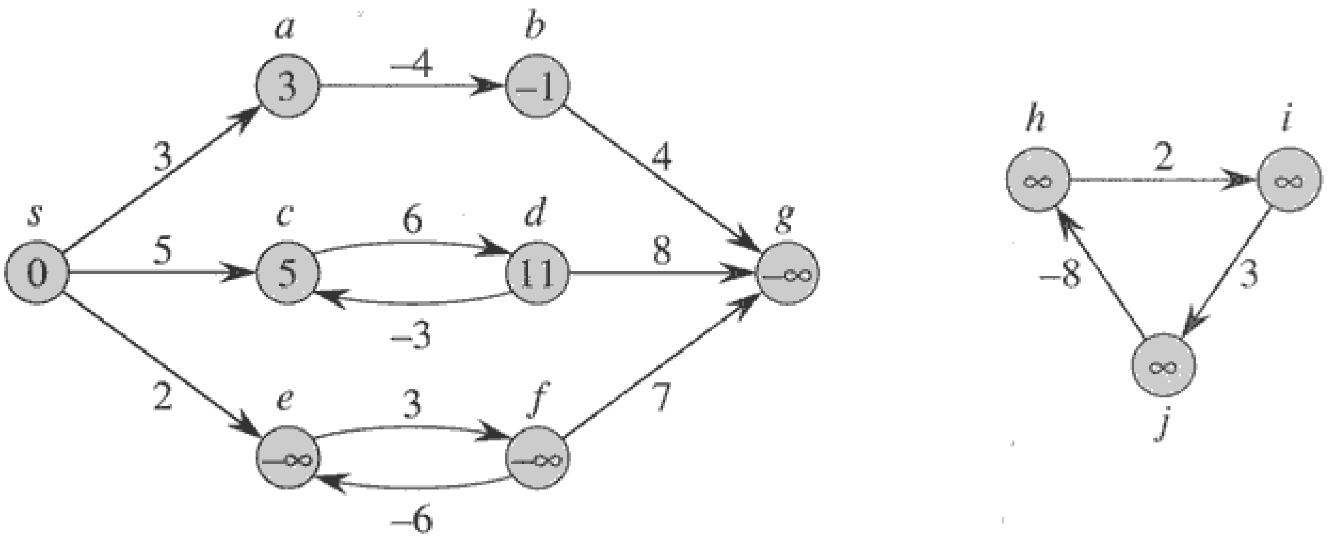
$\delta(s, v) = -\infty$ for all v on the cycle.



Negative-weight edges

- No problem, as long as no negative-weight cycles are reachable from the source.
- Otherwise, we can just keep going around it, and get

$\delta(s, v) = -\infty$ for all v on the cycle.



Initialization

- Maintain $v.d$ for each $v \in V$
- $v.d$ is called *shortest-path weight estimate* and it is *upper bound* on $\delta(s,v)$

INIT(G, s)

for each $v \in V$ do

$v.d \leftarrow \infty$

$v.\pi \leftarrow$ NIL

$s.d \leftarrow 0$

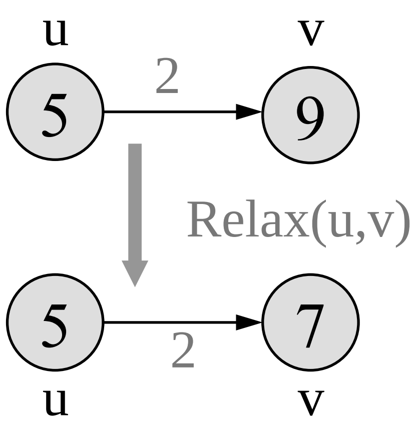
Relaxation

RELAX(u, v)

if v.d > u.d + w(u,v) then

v.d ← u.d + w(u,v)

v. π ← u



Properties of Relaxation

Algorithms differ in

- *how many times* they relax each edge, and
- *the order* in which they relax edges

Question: How many times each edge is relaxed in BFS?

- 0
- $\boxed{1}$
- 2
- More than 2

Content of this Chapter

- ❑ Introduction
- ➢ **The Bellman-Ford algorithm**
- ❑ Single-source shortest paths in DAGs
- ❑ Dijkstra's algorithm

Bellman-Ford Algorithm for Single Source Shortest Paths

- Solves the single-source shortest-paths problem on *weighted*, *directed* graph in the general case (more general than Dijkstra's algorithm):
  - Edge-weights can be negative
- Detects the existence of negative-weight cycle(s) reachable from s.
- The algorithm relaxes edges, progressively decreasing an estimate *v.d* on the weight of a shortest path from the source s to each vertex *v* in V until it achieves the actual shortest-path weight.
- The algorithm returns TRUE *iff* the graph contains no negative-weight cycles that are reachable from the source.

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      v.d = ∞;
       v.π = NIL;
   s.d = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (v.d > u.d + w(u,v))
           return FALSE;
   return TRUE;
```

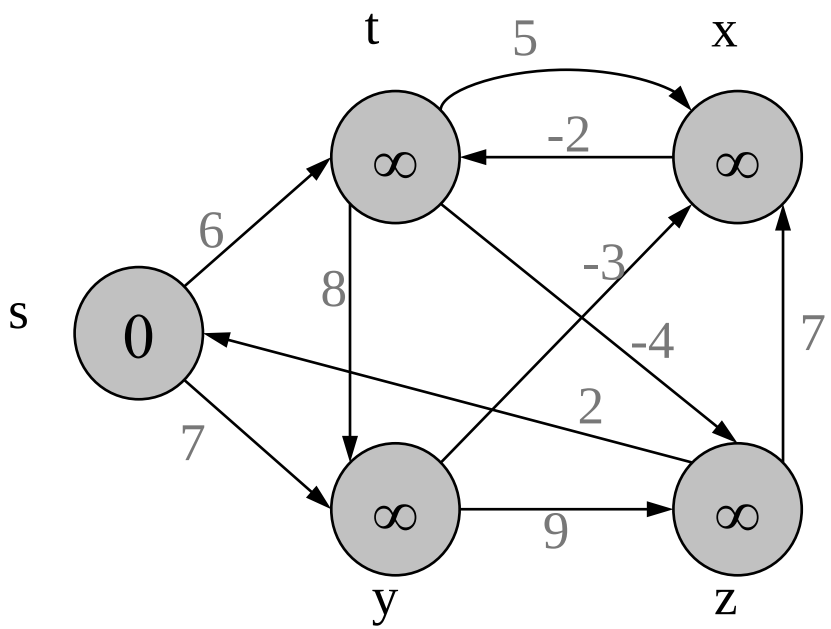
Initialization

Relaxation:
Make |V|-1 passes, relaxing each edge

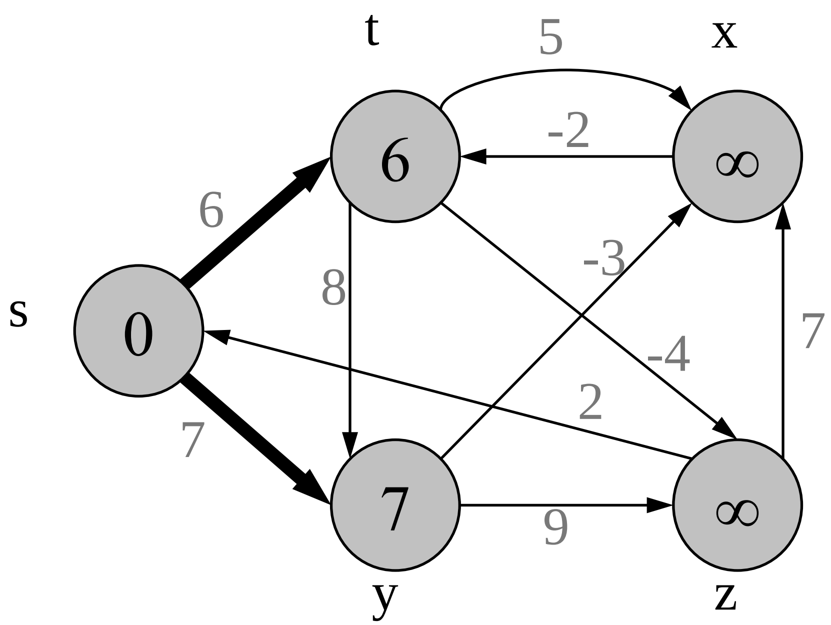
Test for solution
Under what condition do we get a solution?

Relax(u,v,w): if (v.d > u.d + w) then v.d = u.d + w; v.π = u;

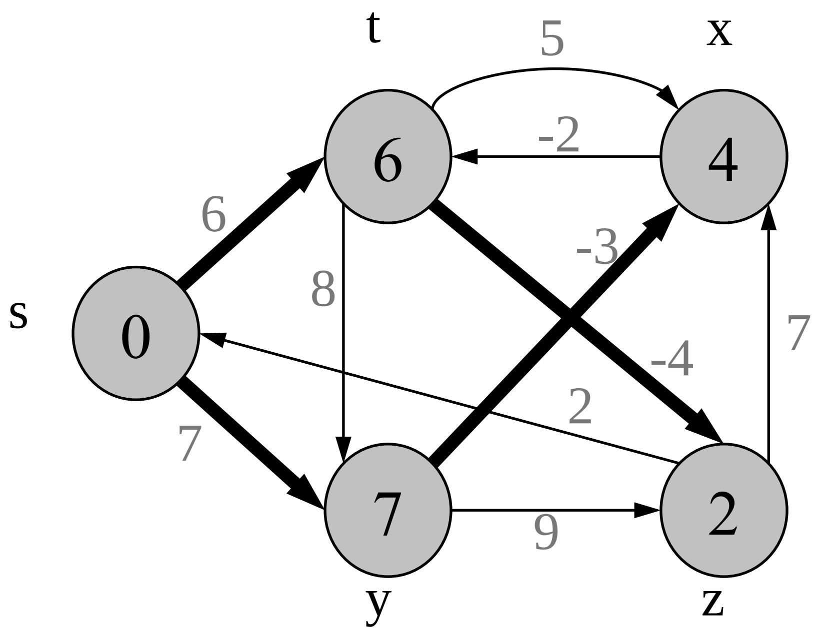
Bellman-Ford Algorithm Example



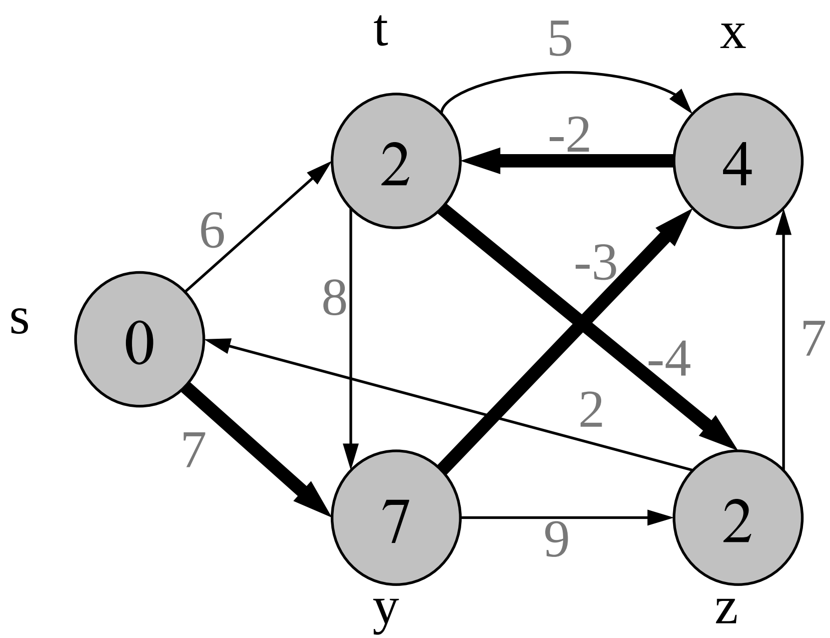
Bellman-Ford Algorithm Example



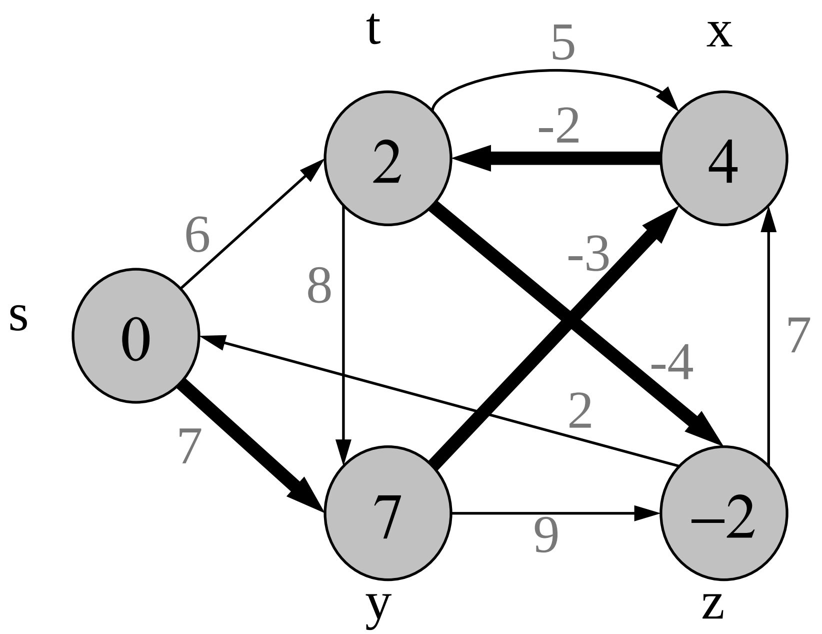
Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Bellman-Ford Algorithm Example



Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      v.d = ∞;
       v.π = NIL;
   s.d = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (v.d > u.d + w(u,v))
           return FALSE;
    return TRUE;
```

What will be the running time?

A: O(VE)

Relax(u,v,w): if (v.d > u.d + w) then v.d = u.d + w; v.π = u;

Bellman-Ford

- Note that order in which edges are processed affects how quickly it converges
- Correctness: show $v.d = \delta(s,v)$ after |V|-1 passes
  - Lemma: v.d ≥ δ(s,v) always
    - Initially true
    - Let v be first vertex for which v.d < δ(s,v)
    - Let u be the vertex that caused v.d to change:
      v.d = u.d + w(u,v)
    - Then v.d < δ(s,v)
      δ(s,v) ≤ δ(s,u) + w(u,v) (*Why?*)
      δ(s,u) + w(u,v) ≤ u.d + w(u,v) (*Why?*)
    - So v.d < u.d + w(u,v). Contradiction.

Bellman-Ford

- Prove: after |V|-1 passes, all *d* values correct
  - Consider shortest path from s to v:
    $s \to v_1 \to v_2 \to v_3 \to v_4 \to v$
    - Initially, s.d = 0 is correct, and doesn't change (*Why?*)
    - After 1 pass through edges, v_1.d is correct (*Why?*) and doesn't change
    - After 2 passes, v_2.d is correct and doesn't change
    - …
    - Terminates in |V| - 1 passes: (*Why?*)
    - *What if it doesn't?*

Content of this Chapter

- Introduction
- The Bellman-Ford algorithm
- **Single-source shortest paths in DAGs**
- Dijkstra’s algorithm

Single-Source Shortest Paths in DAGs

- Shortest paths are always *well-defined* in *dags*
  - no cycles => no negative-weight cycles even if there are negative-weight edges
- **Idea:**
  - To process vertices on each shortest path from left to right, we would be done in 1 pass due to *L4*.

Single-Source Shortest Paths in DAGs

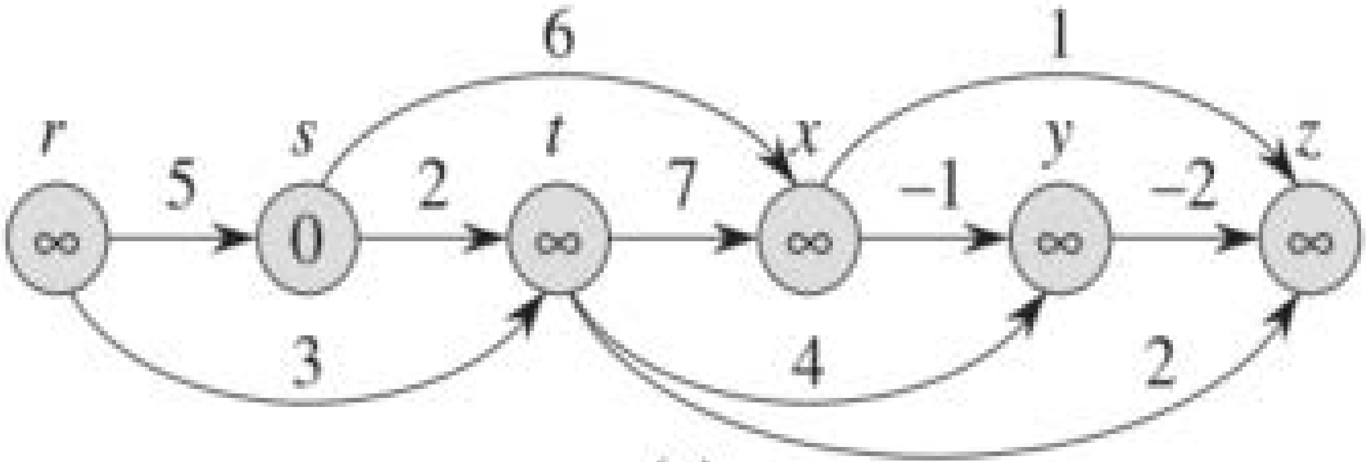
In a dag:

- Every path is a subsequence of the topologically sorted vertex order.
- **We will process each path in forward order.**
  - **Never relax edges out of a vertex until have processed all edges into the vertex.**
- Thus, just 1 pass is sufficient.

Single-Source Shortest Paths in DAGs

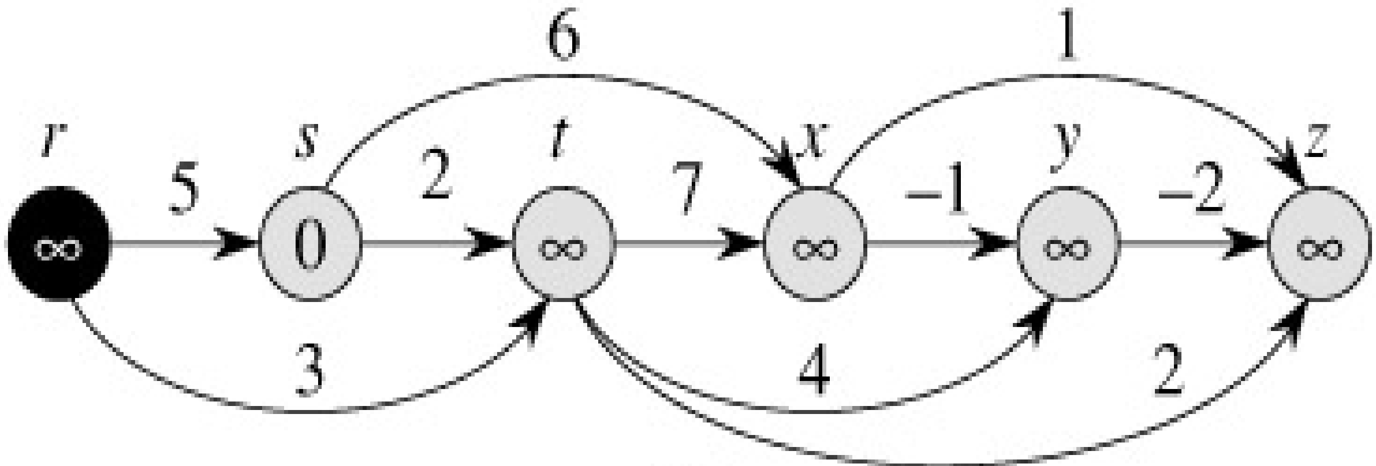
```
DAG-SHORTEST PATHS(G, s)
    TOPOLOGICALLY-SORT the vertices of G
    INIT(G, s)
    for each vertex u taken in topologically sorted order do
        for each v ∈ Adj[u] do
            RELAX(u, v)
```

Example



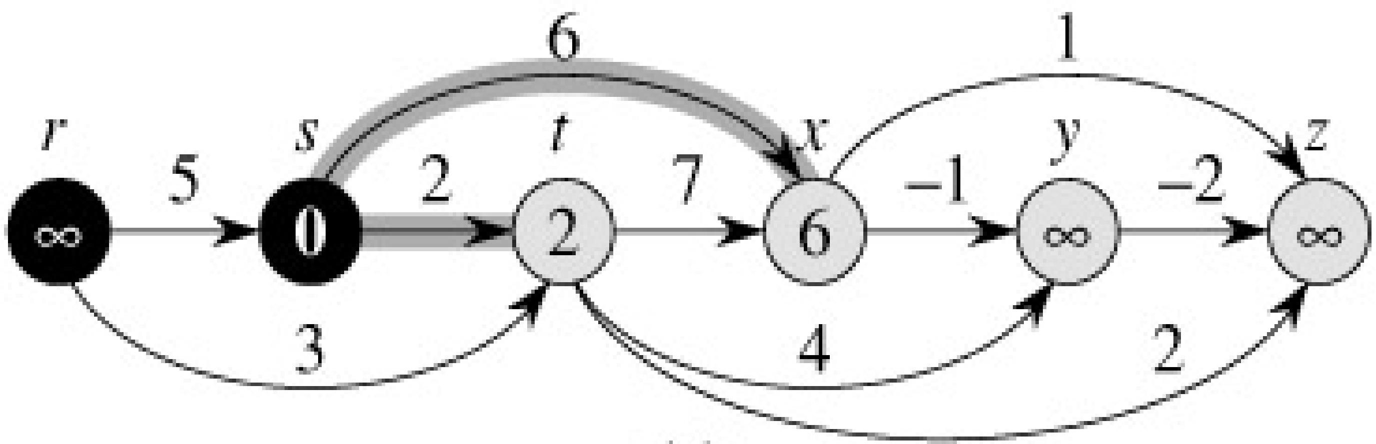
(a)

Example



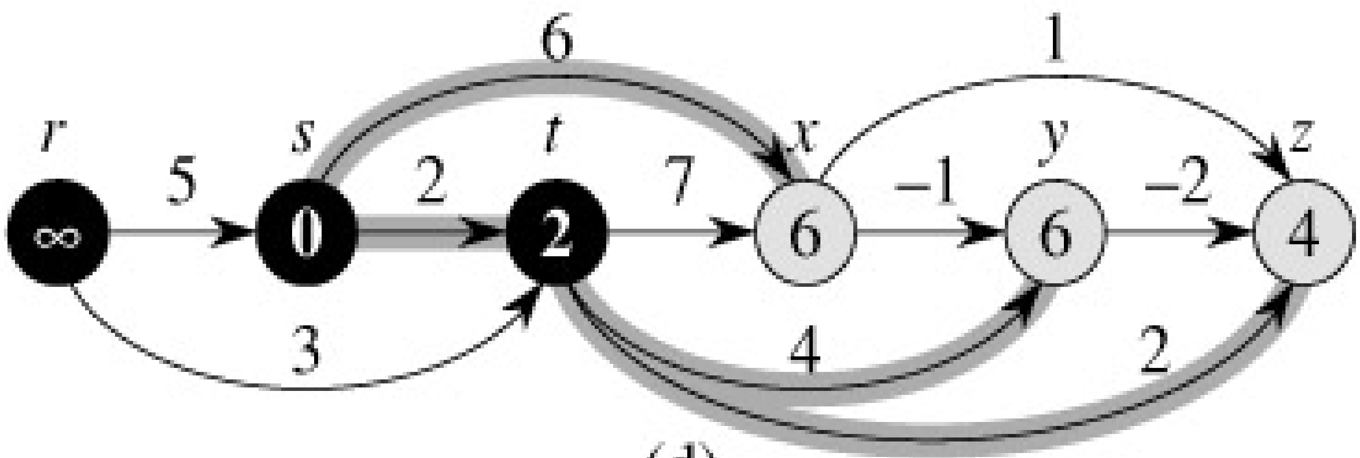
(b)

Example



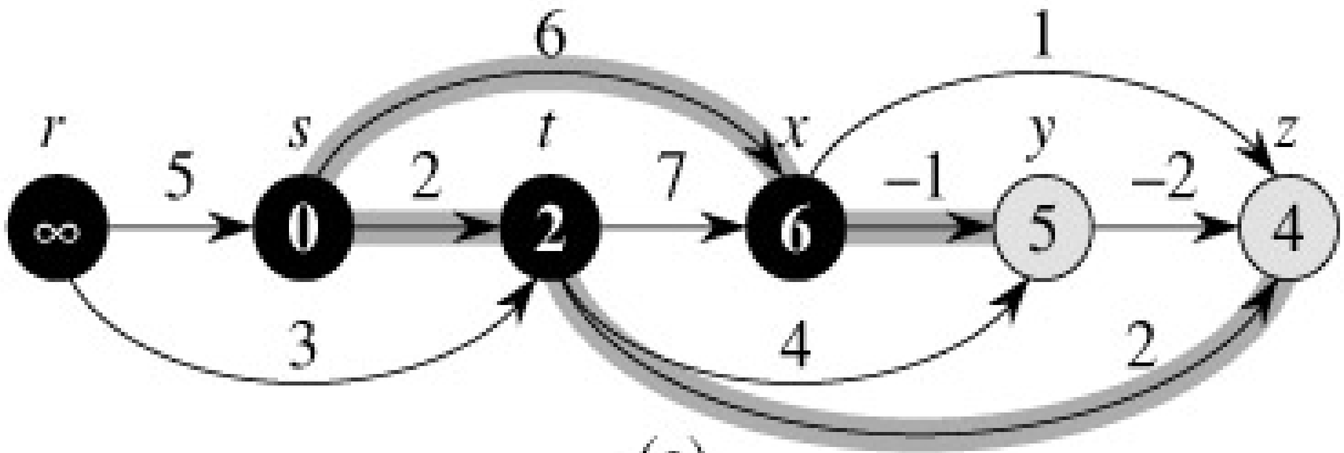
(c)

Example



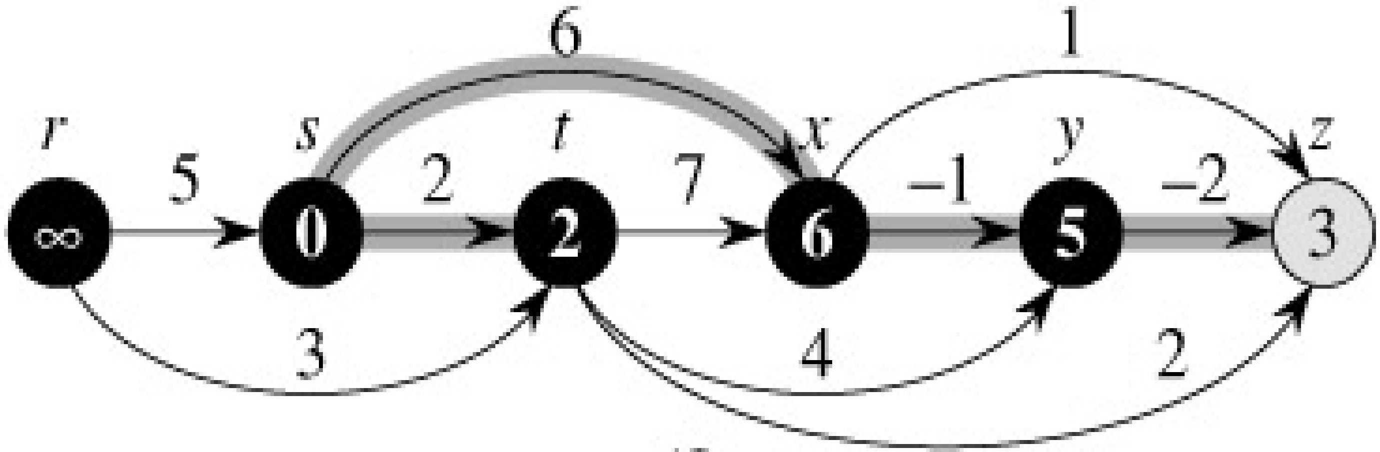
(d)

Example



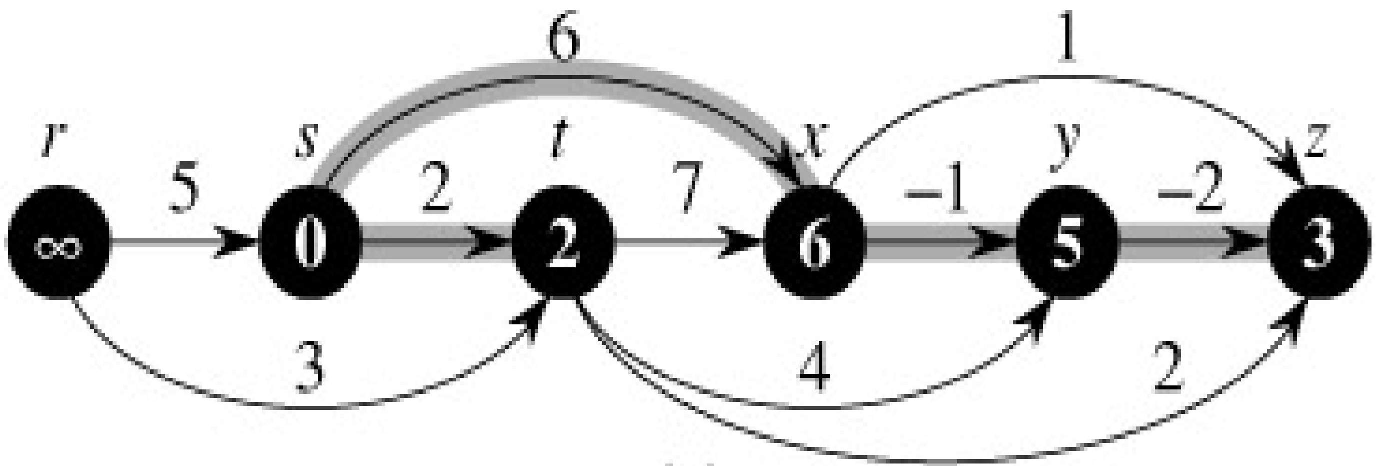
(e)

Example



(f)

Example



(g)

Single-Source Shortest Paths in DAGs

Runs in linear time: $\Theta(V+E)$ because:

- Topological sort: $\Theta(V+E)$
- Initialization: $\Theta(V)$
- *for-loop:* $\Theta(V+E)$
  - Each vertex processed exactly once
  - Each edge processed exactly once

Content of this Chapter

- Introduction
- The Bellman-Ford algorithm
- Single-source shortest paths in DAGs
- **Dijkstra's algorithm**

Dijkstra's Algorithm

- Solves the single-source shortest-paths problem on *weighted*, *directed* graph with *no negative edge weights*.
- Similar to breadth-first search
  - Grows a tree gradually, advancing from vertices taken from a queue.
- Also similar to Prim's algorithm for MST
  - Uses a min-priority queue keyed on $v.d$

Dijkstra's Algorithm

- Dijkstra's algorithm maintains a set *S* of vertices whose final shortest-path weights from the source *s* have already been determined.

- The algorithm repeatedly selects the vertex *u* in *V*-*S* with the minimum shortest-path estimate, adds *u* to *S*, and relaxes all edges leaving *u*.

Dijkstra's Algorithm

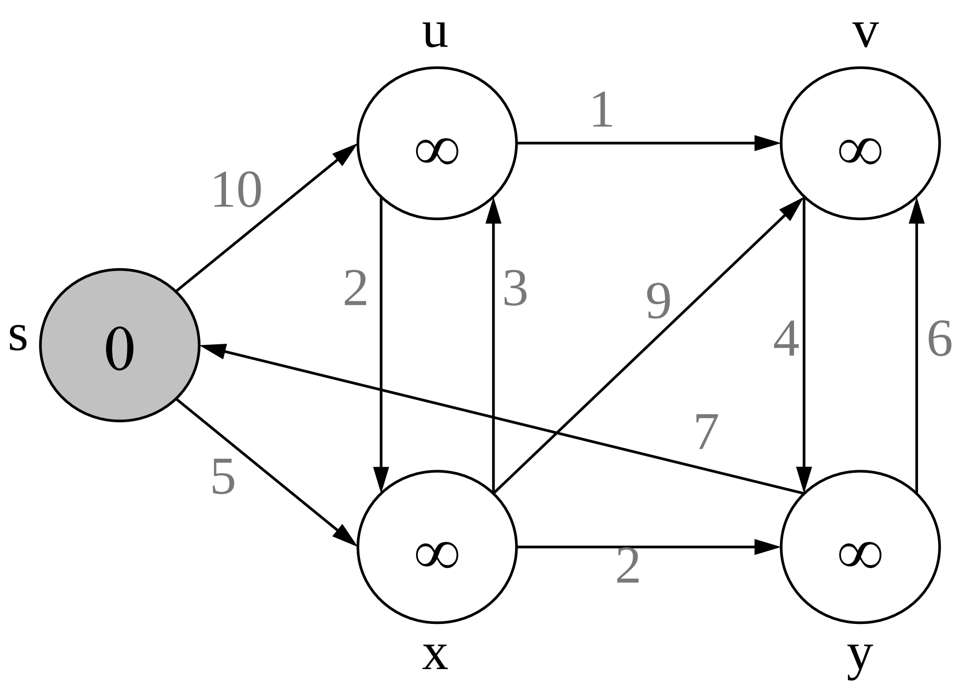
```
Dijkstra(G,s)
   for each v ∈ V
      v.d = ∞; v.π = NIL;
   s.d = 0; S = ∅;
   Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S U {u};
      for each v ∈ Adj[u]
         if (v.d > u.d + w(u,v))
            v.d = u.d + w(u,v);
            v.π = u;
```

Relaxation Step

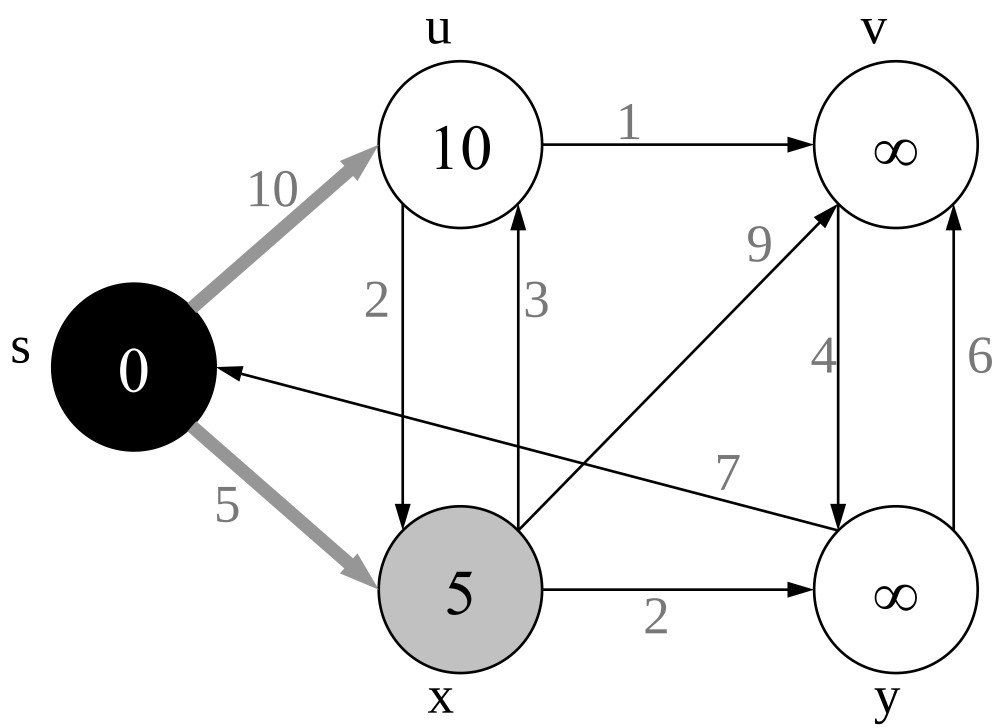
Relaxation Step

DecraseKey()

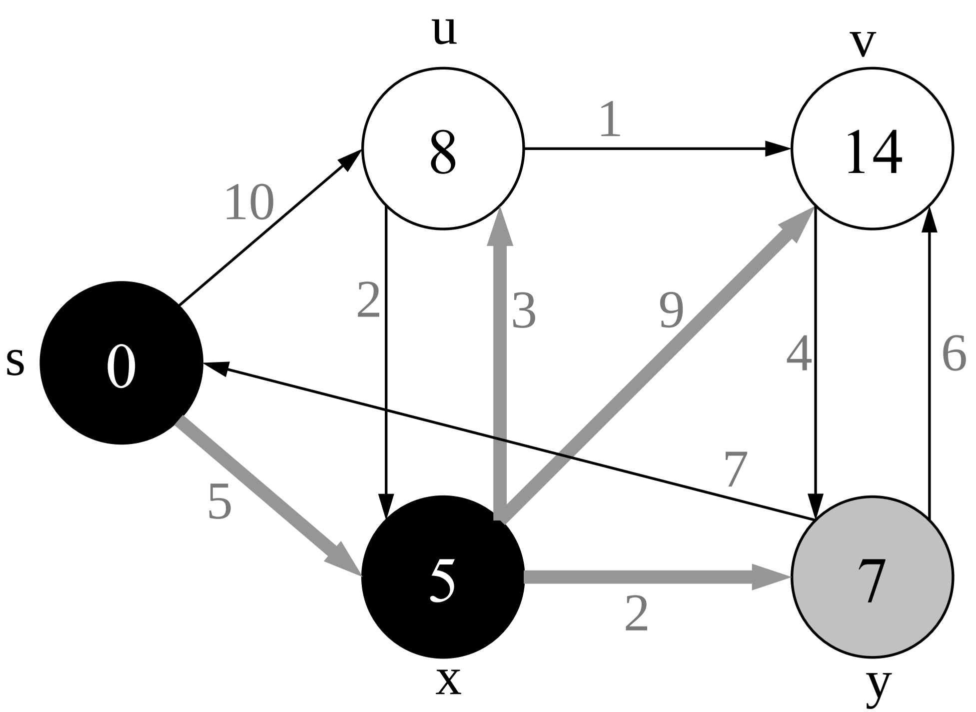
Example



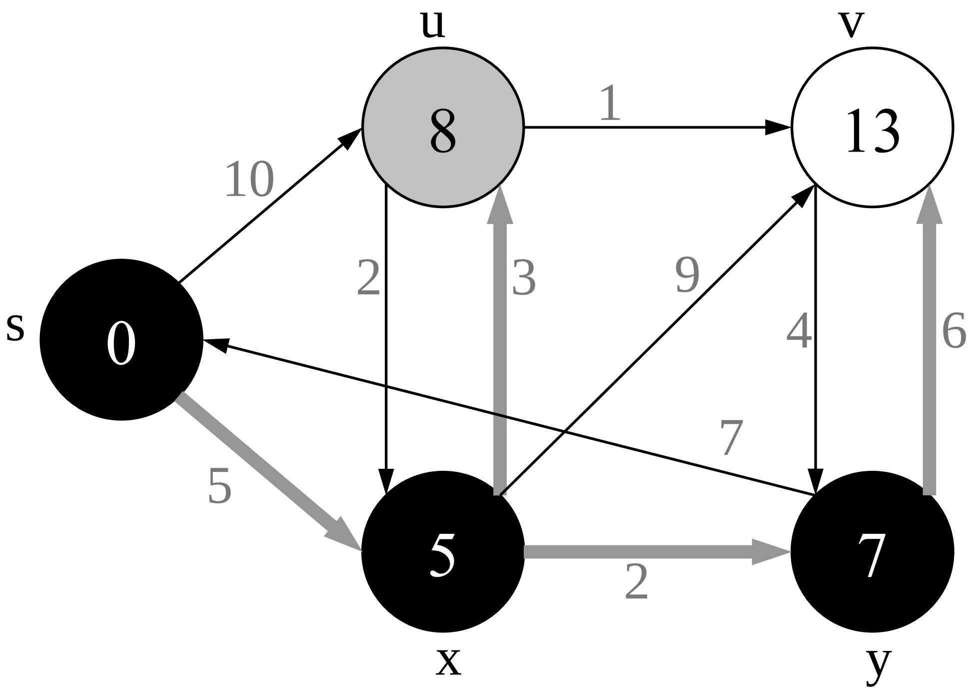
Example



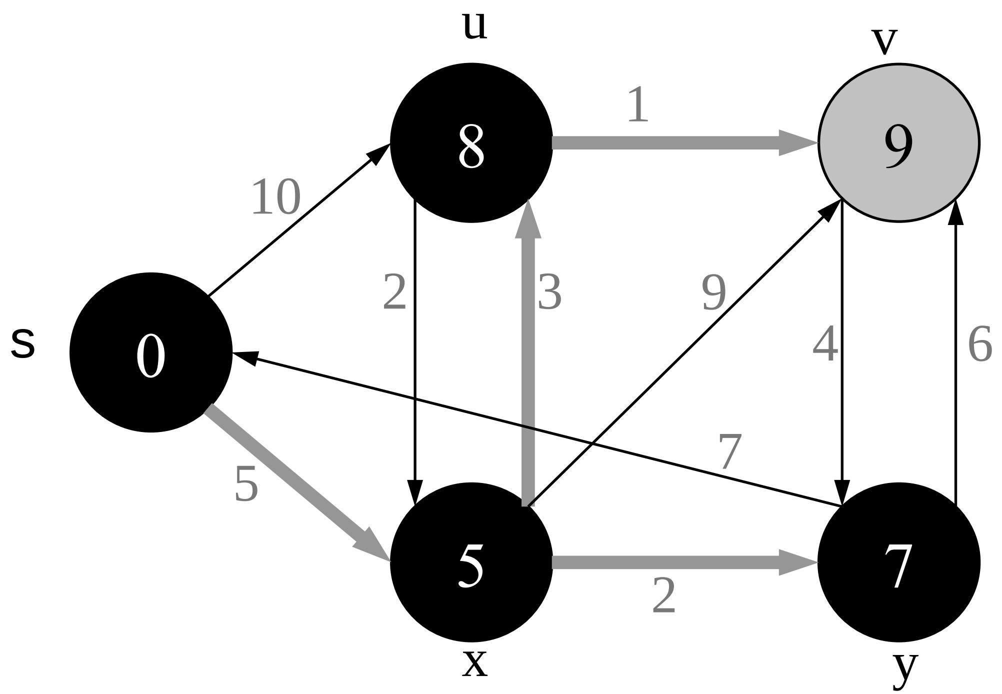
Example



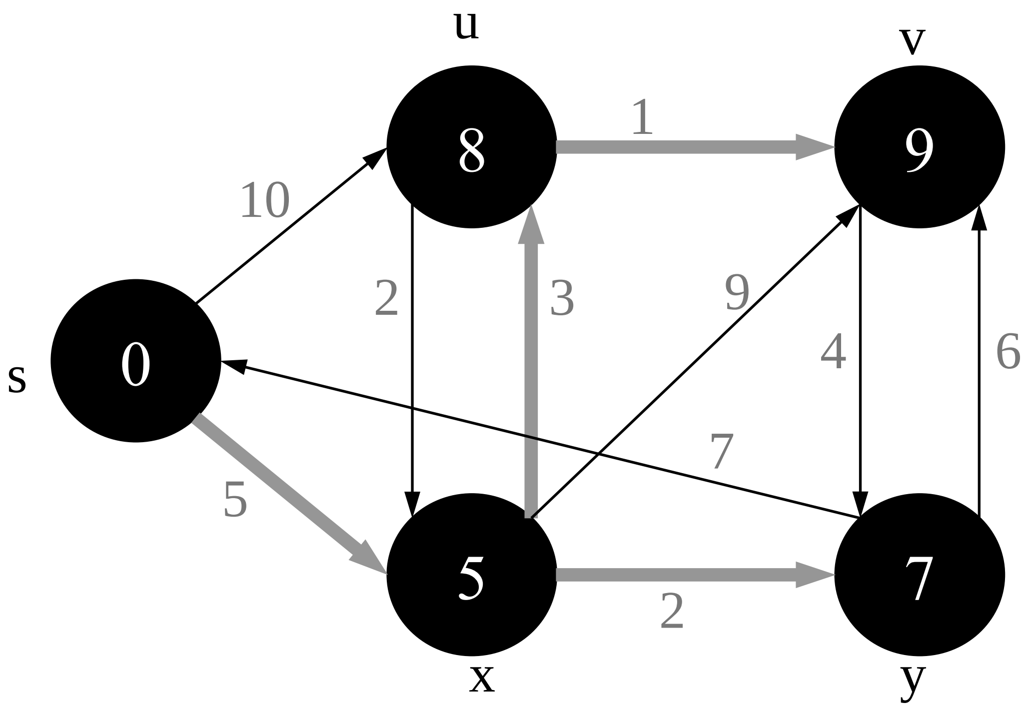
Example



Example



Example



Dijkstra's Algorithm Runtime

```
Dijkstra(G)
   for each v ∈ V
      v.d = ∞; v.π = NIL;
   s.d = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S U {u};
      for each v ∈ Adj[u]
         if (v.d > u.d+w(u,v))
            v.d = u.d+w(u,v);
            v.π = u;
```

How many times is ExtractMin() called? **O(V)**

How many times is DecraseKey() called? **O(E)**

What will be the total running time? **O(E lg V)**