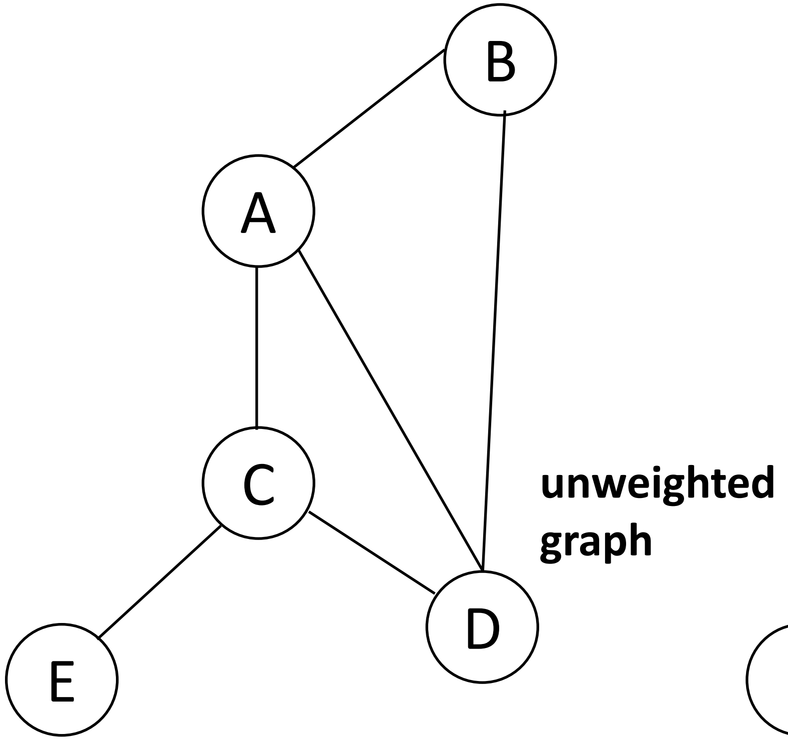
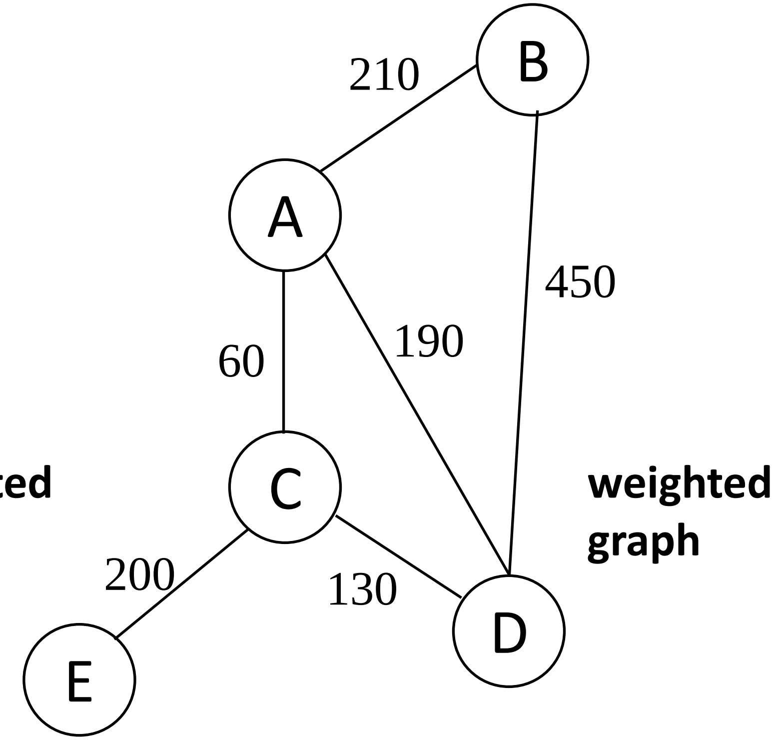
Shortest Path Problem

Bellman Ford Algorithm

Shortest Path Problems

- **What is shortest path ?**
  - <u>shortest length between two vertices</u> for an unweighted graph:
  - <u>smallest cost between two vertices</u> *for* a weighted graph:

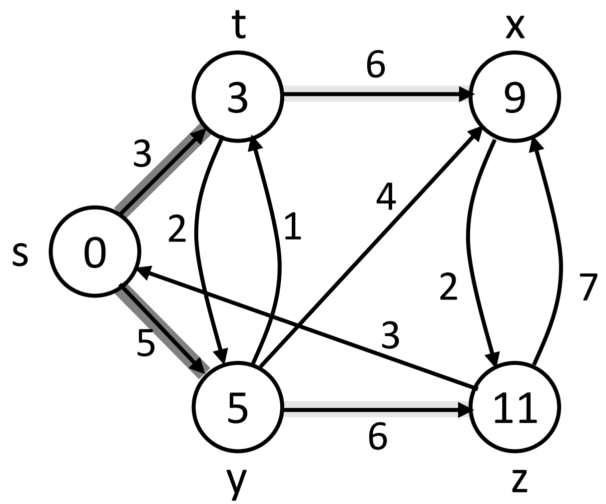




Shortest Path Problems

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**
- **Weight of path** p = $\langle v_0, v_1, \ldots, v_k \rangle$

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



- **Shortest-path weight** from u to v:

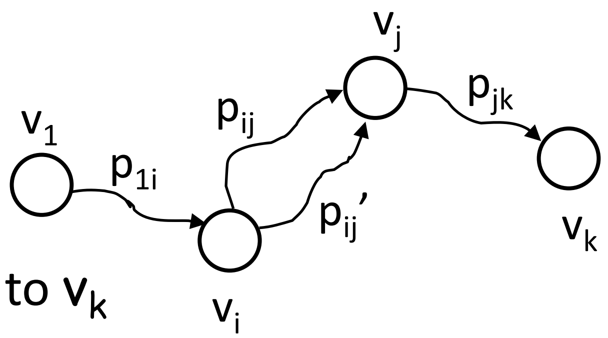
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

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

Variants of Shortest Paths

- **Single-source shortest path**
  - G = (V, E) $\Rightarrow$ find a shortest path from a given source vertex **s** to each vertex v $\in$ V
- **Single-destination shortest path**
  - Find a shortest path to a given destination vertex **t** from each vertex v
  - Reverse the direction of each edge $\Rightarrow$ single-source
- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
  - Solve the single-source problem
- **All-pairs shortest-paths**
  - Find a shortest path from u to v for every pair of vertices u and v

Optimal Substructure of Shortest Paths



Given:

- A weighted, directed graph $G = (V, E)$
- A weight function $w: E \rightarrow \mathbf{R}$,
- A shortest path $p_{1k} = \langle v_1, v_2, \ldots, v_k \rangle$ from v_1 to v_k
- A subpath of p: $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, with $1 \leq i \leq j \leq k$

Then: p_{ij} is a shortest path from v_i to v_j

Proof: $p = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$

Assume $\exists\, p_{ij}'$ from v_i to v_j with $w(p_{ij}') < w(p_{ij})$

$\Rightarrow w(p') = w(p_{1i}) + w(p_{ij}') + w(p_{jk}) < w(p)$ contradiction!

Using BFS

Generalization of BFS to handle weighted graphs

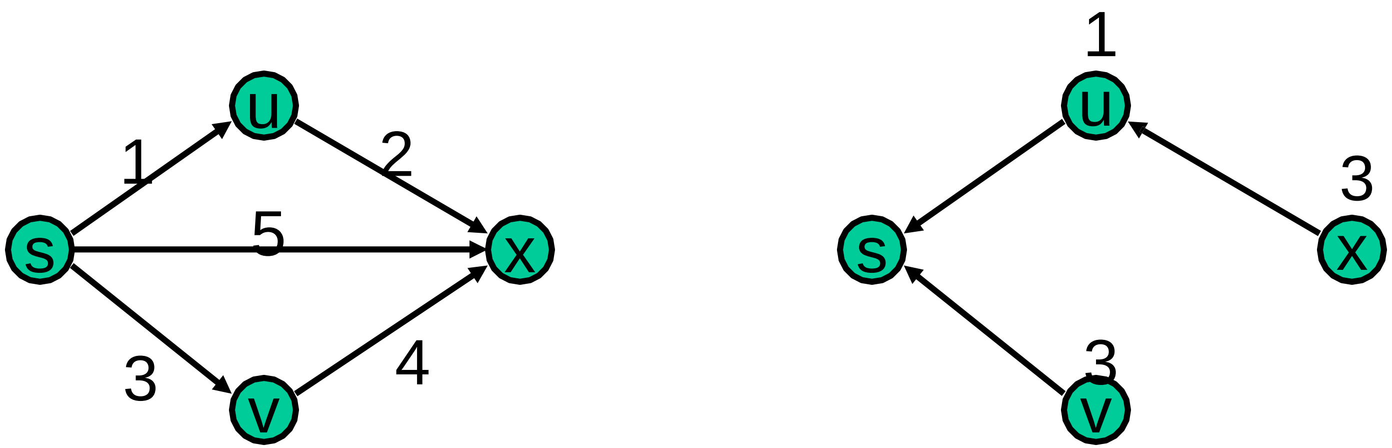
- Direct Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, edge weight *f*n ; w : $\boldsymbol{E} \rightarrow \boldsymbol{R}$
- In BFS w(e)=1 for all e $\in$ E

Weight of path p = $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

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

Single Source Shortest Path Problem

- Given a graph and a start vertex s
  - Determine distance of every vertex from s
  - Identify shortest paths to each vertex
    - Express concisely as a “shortest paths tree”
    - Each vertex has a pointer to a predecessor on shortest path



Triangle Inequality

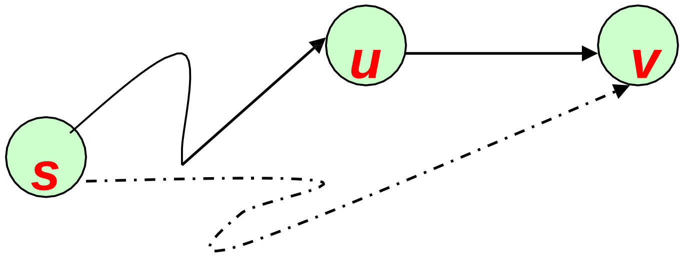
Lemma 1: for a given vertex $s \in \boldsymbol{V}$ *and for every edge* $(u,v) \in E$,

- $\delta(s,v) \leq \delta(s,u) + w(u,v)$

Proof: shortest path $s \rightsquigarrow v$ *is no longer than any other path.*

- in particular the path that takes the shortest path $s \rightsquigarrow v$ *and*

then takes cycle (u,v)



Initialization

Alg.: INIT (G, s)

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

2. **do** $d[v] \leftarrow \infty$

3. $\pi[v] \leftarrow$ NIL

4. $d[s] \leftarrow 0$

- All the shortest-paths algorithms start with INIT
- Maintain $d[v]$ for each $v \in V$
- $d[v]$ is called the shortest-path weight estimate and it is the upper bound on $\delta(s, v)$

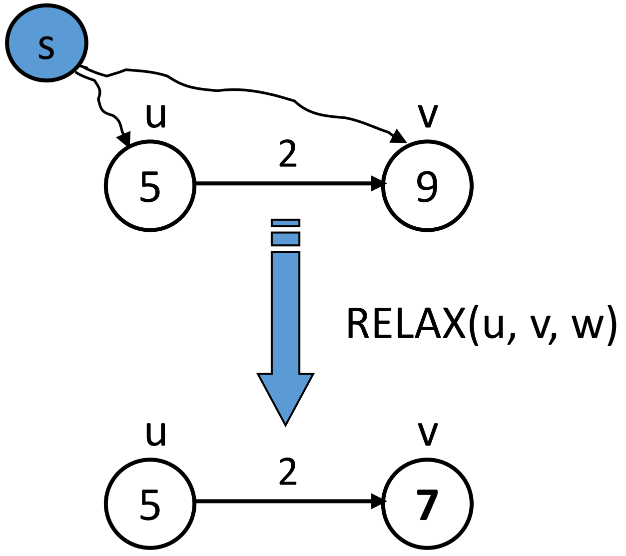
Relaxation

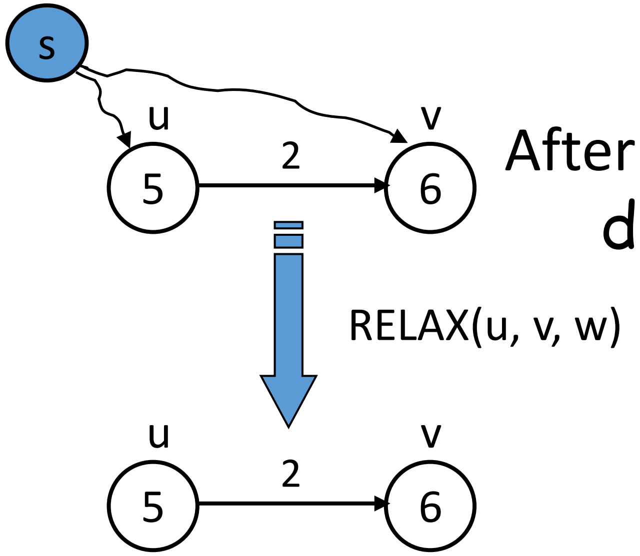
- **Relaxing** an edge (u, v) = testing whether we can improve the shortest path to v found so far by going through u

  If $d[v] > d[u] + w(u, v)$

  we can improve the shortest path to v

  $\Rightarrow$ update $d[v]$ and $\pi[v]$





After relaxation:

$$d[v] \leq d[u] + w(u, v)$$

RELAX(u, v)

1. **if** d[v] > d[u] + w(u, v)
2. **then** d[v] $\leftarrow$ d[u] + w(u, v)
3. π[v] $\leftarrow$ u

- All the single-source shortest-paths algorithms
  - start by calling INIT
  - then relax edges
- The algorithms differ in the order and how many times they relax each edge (it affects the correctness and time complexity of the algorithm)

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?

Answer: Only once!

Properties of Relaxation

Given:

- An edge weighted directed graph $G = (V, E)$ with *edge weight function* $(w{:}E \to R)$ and a source vertex $s \in V$
- G is initialized by ***INIT(G , s)***

Lemma 2: Immediately after relaxing edge (u,v),

$$d[v] \leq d[u] + w(u,v)$$

Lemma 3: For any sequence of relaxation steps over E,

(a) the invariant $d[v] \geq \delta(s,v)$ is maintained

(b) once d[v] achieves its lower bound, it never changes.

Properties of Relaxation

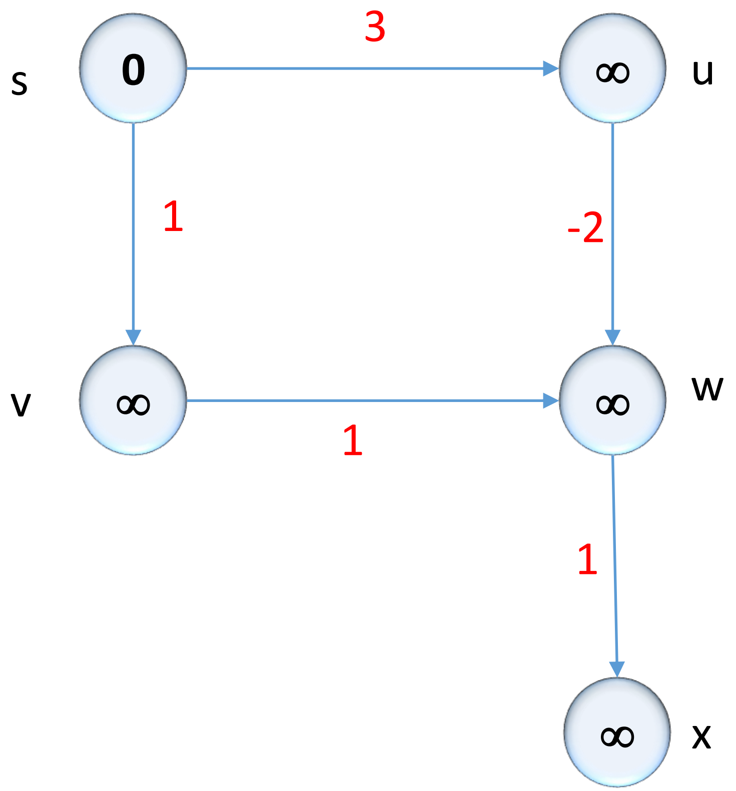
Lemma 4: Let $s \rightsquigarrow u \rightarrow v$ *be a shortest path from s to v for some* $u, v \in V$

- Suppose that a sequence of relaxations including ***RELAX(u,v)*** were performed on E
- If $d[u] = \delta(s, u)$ at any time prior to ***RELAX(u, v)***
- then $d[v] = \delta(s, v)$ at all times after ***RELAX(u, v)***

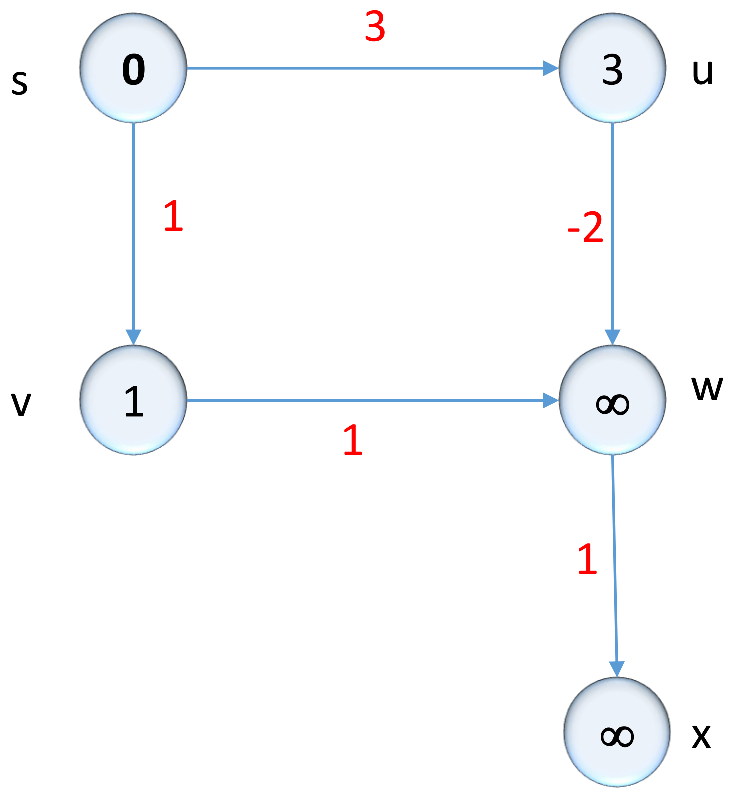
Dijkstra’s Algorithm For Shortest Paths

```
DIJKSTRA(G, s)
    INIT(G, s)
    S←Ø                 > set of discovered nodes
    Q←V[G]
    while Q ≠Ø do
        u←EXTRACT-MIN(Q)
        S←S U {u}
        for each v ∈ Adj[u] do
            RELAX(u, v)  > May cause
                         > DECREASE-KEY(Q, v, d[v])
```

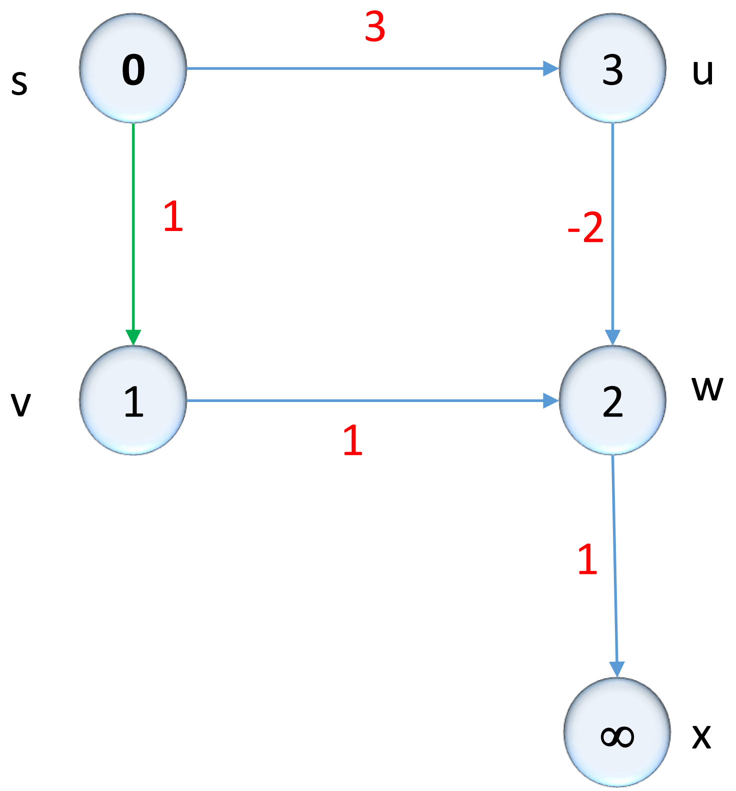
Negative Edges



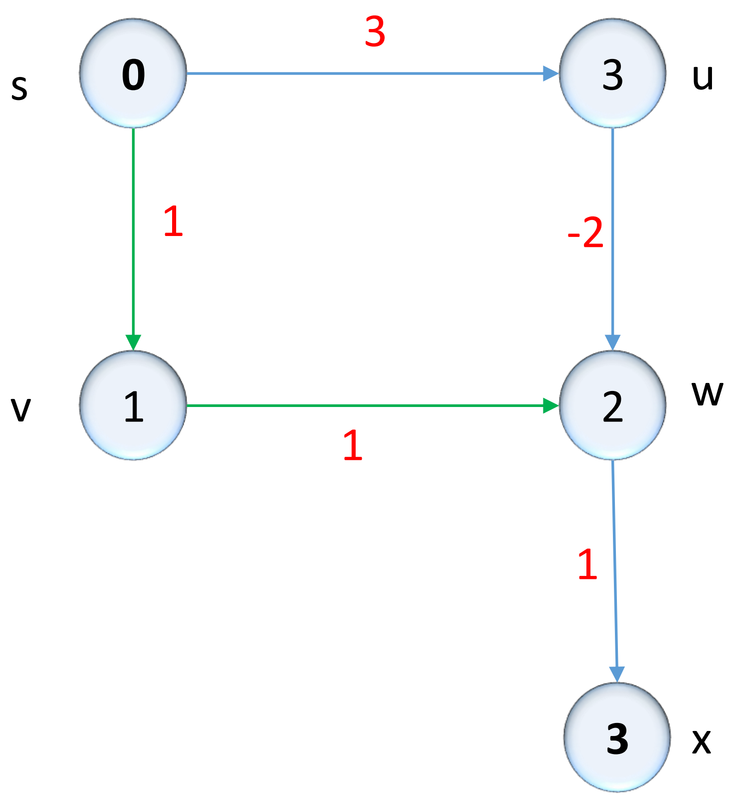
Negative Edges



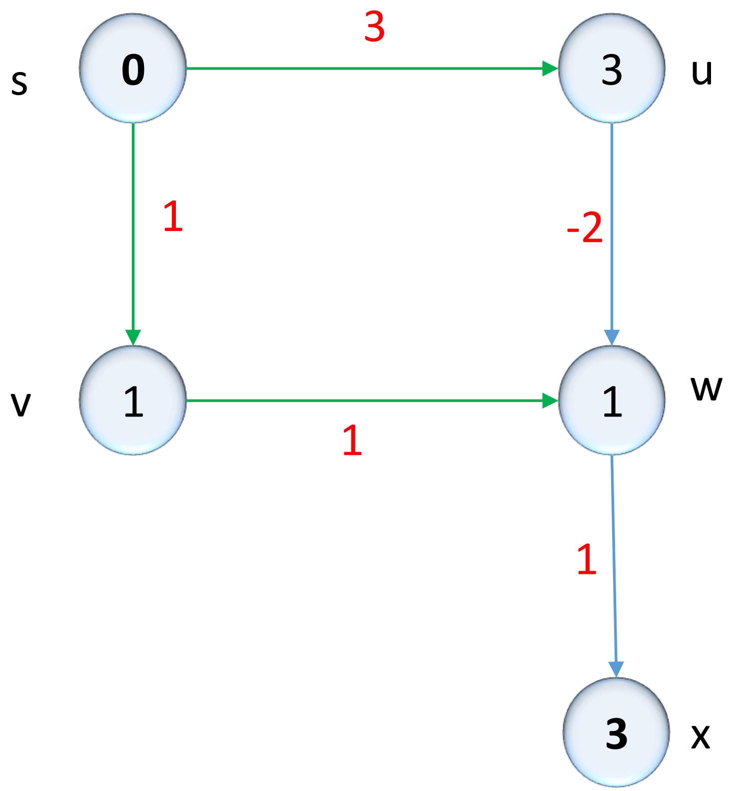
Negative Edges



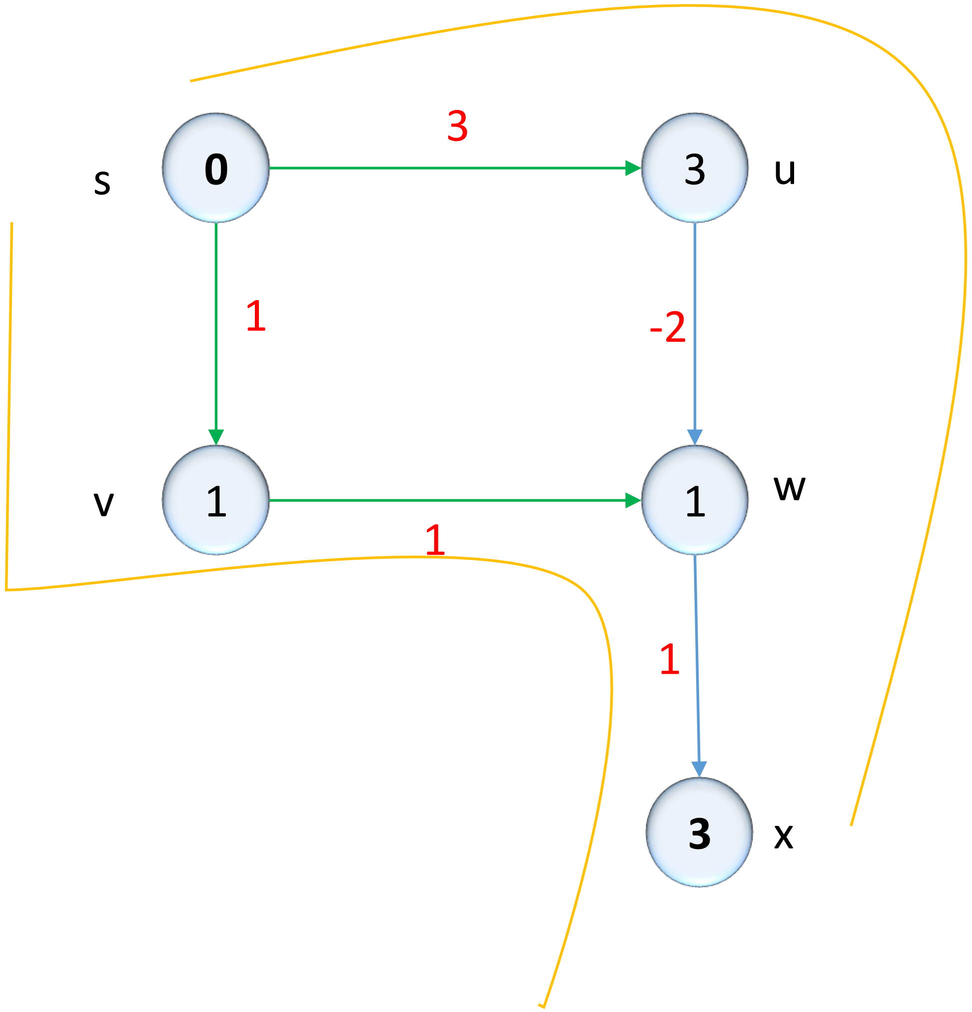
Negative Edges



Negative Edges



Negative Edges



Bellman Ford

- Dijkstra’s SSSP algorithm requires all edge weights to be non-negative.

- Bellman Ford Algorithm can handle negative weight edges. It can even detect negative cycles.

Bellman Ford – Basic Idea

- Consider each edge (u,v) and see if u offers v a cheaper path from s
  - compare d[v] to d[u] + w(u,v)
- Repeat this process |V| - 1 times to ensure that accurate information propagates from s, no matter what order the edges are considered in

Bellman Ford Algorithm

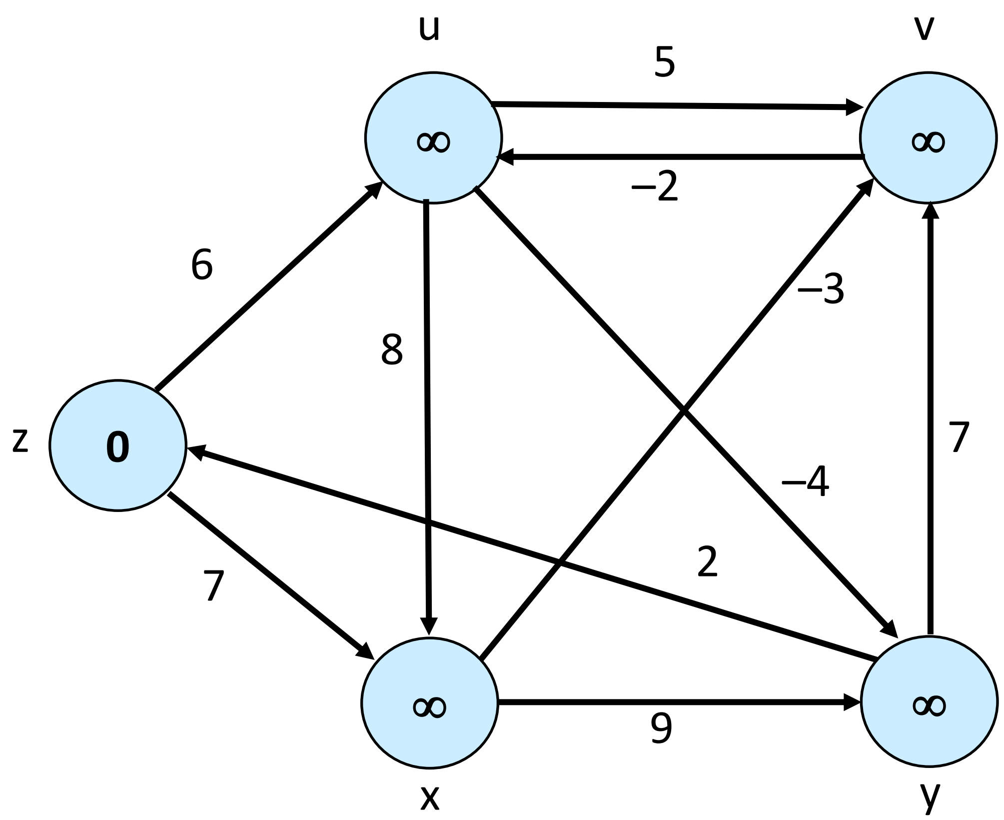
- INIT(G, s);
- **for** i := 1 to |V[G]| –1 **do**
-     **for** each (u, v) in E[G] **do**
-         Relax(u, v)
- **for** each (u, v) in E[G] **do**
-     **if** d[v] > d[u] + w(u, v) **then**
-         **return** false
- **return** true

Time Complexity is O(VE).

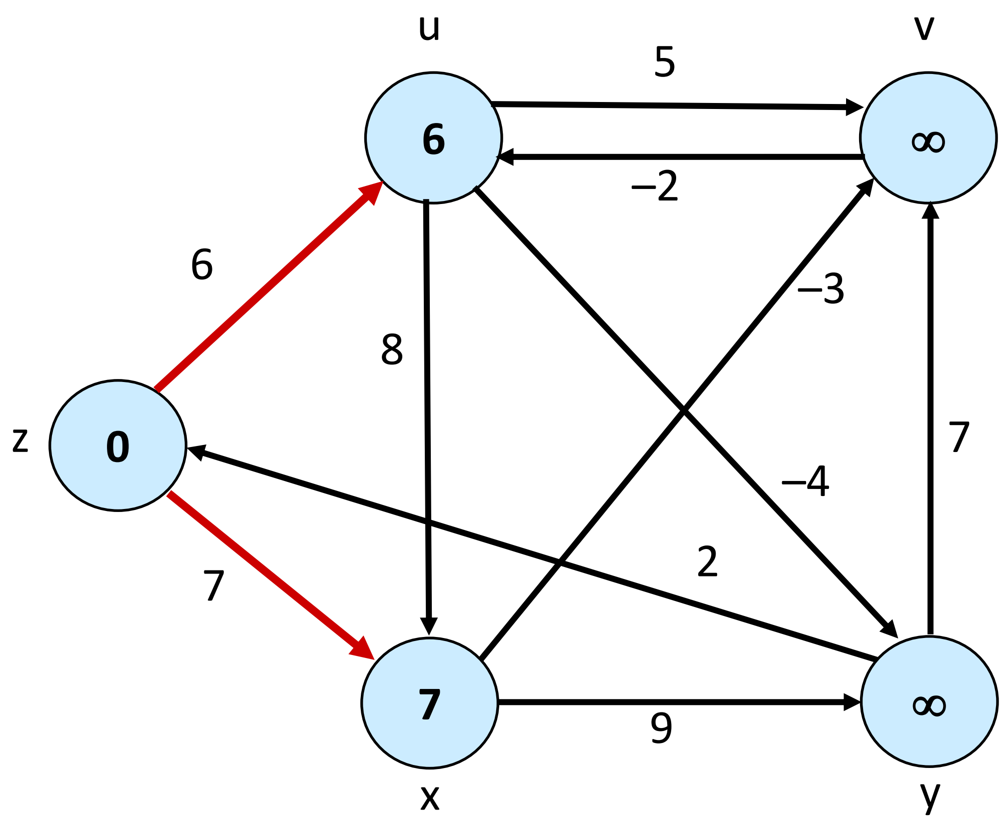
Bellman Ford Algorithm

- First nested for-loop performs |V|-1 relaxation passes; relax every edge at each pass
- Last for-loop checks the existence of a negative-weight cycle reachable from s
  - So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

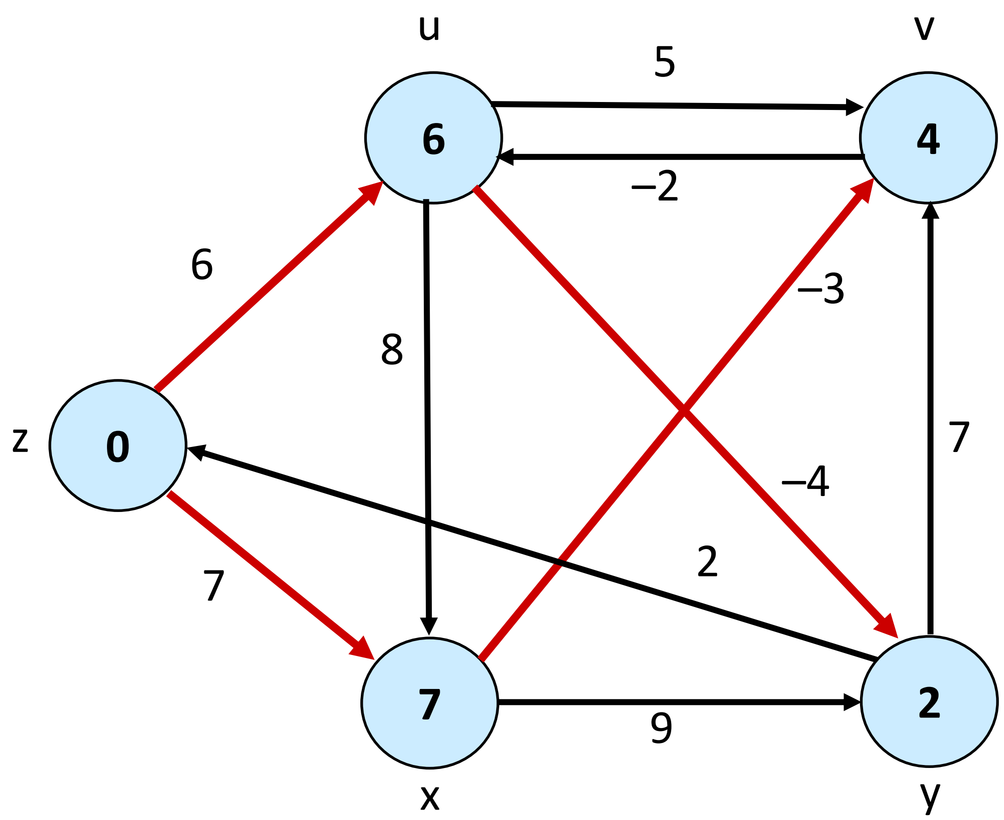
Example



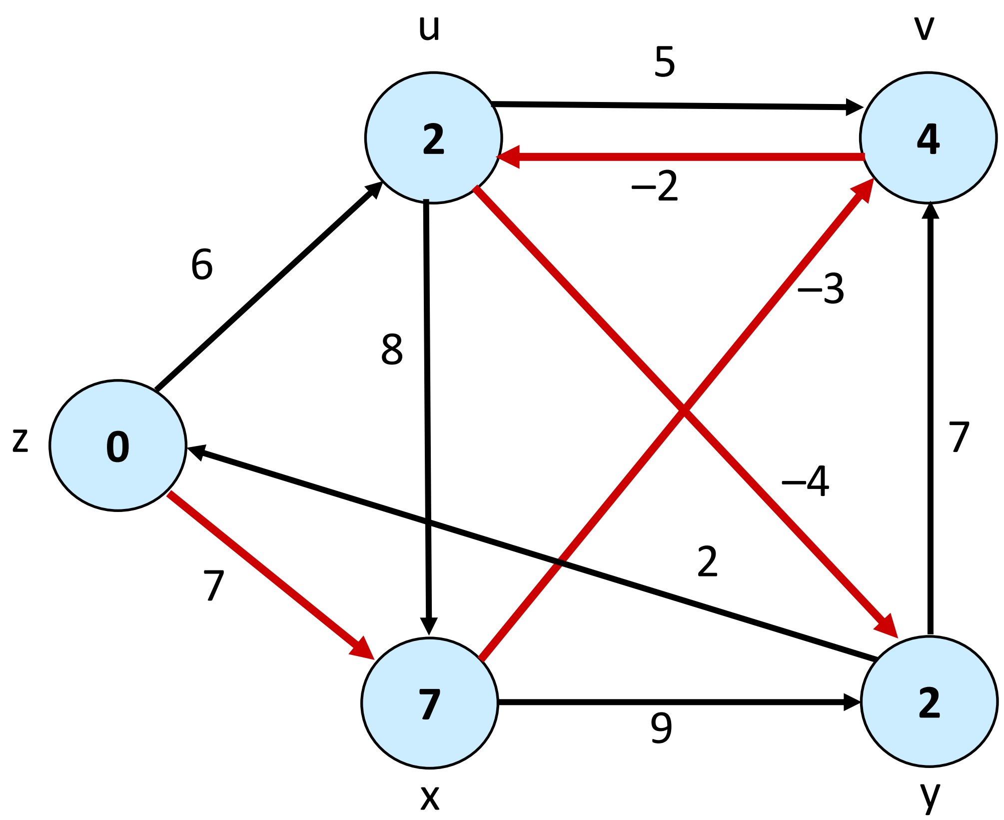
Example



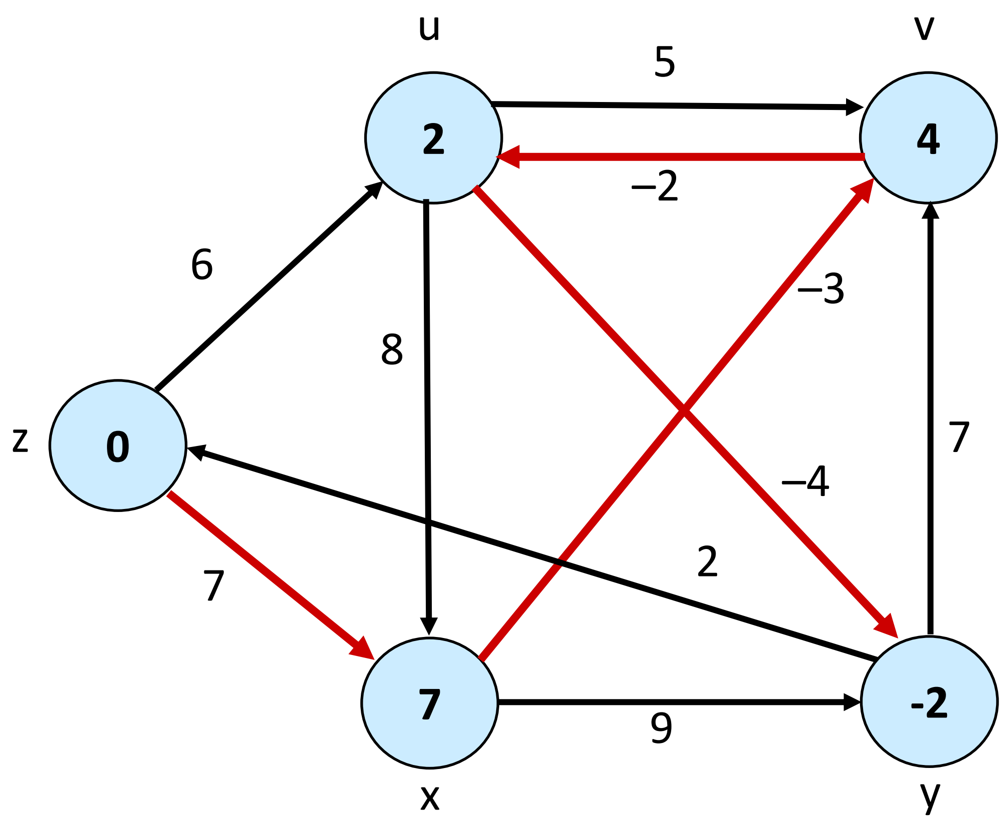
Example



Example



Example



Example

- Converges in just 2 relaxation passes
- Values you get on each pass & how early converges depend on edge process order
- d value of a vertex may be updated more than once in a pass

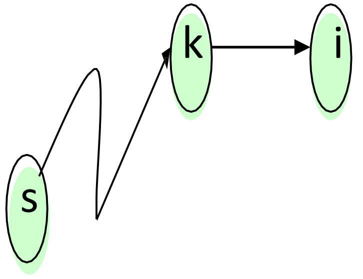
Another Look

Note: This is essentially **dynamic programming**.

d(i, j) is the cost of the shortest path from s to vertex i with at most j edges in it (i.e. number of hops)

Let d(i, j) = cost of the shortest path from s to i that is at most j hops.

$$d(i, j) = \begin{cases} 0 & \textbf{if } i = s \wedge j = 0 \\ \infty & \textbf{if } i \neq s \wedge j = 0 \\ \min(\{d(k, j-1) + w(k, i) : i \in Adj(k)\} \cup \{d(i, j-1)\}) & \textbf{if } j > 0 \end{cases}$$



j ↓ \ i →	z 1	u 2	v 3	x 4	y 5
0	0	∞	∞	∞	∞
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	–2