Properties of Shortest Paths and Relaxation

- Triangle inequality (Lemma 24.10)
- Upper-bound property (Lemma 24.11)
- No-path property (Corollary 24.12)
- Convergence property (Lemma 24.14)
- Path-relaxation property (Lemma 24.15)
- Predecessor-subgraph property (Lemma 24.17)

Triangle Inequality (Lemma 24.10)

- Let G=(V,E) be a weighted, directed graph with weight function w : E→R and source vertex s. Then, for all edges (u,v)∈E, we have δ(s,v)≤δ(s,u)+w(u,v)
- Proof:
  - Suppose that there is a shortest path p from source s to v. Then p has no more weight than any other path from s to v. Specifically, path p has no more weight than the particular path that takes a shortest path from source s to vertex u and then takes edge (u,v).
  - Otherwise, (there is no shortest path from s to v). Exercise 24.5-3.

Upper-bound Property (Lemma 24.11)

- Let
  - G=(V,E) be a weighted, directed graph with weight function $w : E \rightarrow R$
  - $s \in V$ the source vertex
- The graph G is initialized by INITILIZE-SINGLE-SOURCE(G,s).
- Then, we have $v.d \geq \delta(s,v)$ for all $v \in V$. This invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once v.d achieves its lower bound $\delta(s,v)$, it never changes.

Upper-bound Property (Lemma 24.11)

- Proof by induction over the number of relaxation steps
  - Basis case (0 relaxation)
    - $v.d = \infty \geq \delta(s,v)$ for all vertices $v \in V-\{s\}$
    - $s.d = 0 = \delta(s,s)$

Upper-bound Property (Lemma 24.11)

- Proof by induction over the number of relaxation steps
  - Induction step
    - Induction hypothesis: $v.d \geq \delta(s,v)$ for all $v \in V$ prior to the relaxation of an edge (u,v)
    - The only d value that may change is v.d. If it changes, we have
      $v.d = u.d + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$
  - To see that the value of v.d never changes once $v.d = \delta(s,v)$
    - v.d cannot decrease because $v.d \geq \delta(s,v)$
    - v.d cannot increase because relaxation steps do not increase d values

No-path Property (Corollary 24.12)

- Suppose that in a weighted, directed graph $G=(V,E)$ with weight function $w:E \rightarrow R$, no path connects a source s to a given vertex v.
- Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(G,s), we have $v.d = \delta(s,v) = \infty$ and this equality is maintained as an invariant over any sequence of relaxation steps on the edges of G.
- Proof: By the upper-bound property, we always have $\infty = \delta(s,v) \leq v.d$ and thus we have $v.d = \infty = \delta(s,v)$.

Lemma 24.13

- Let G=(V,E) be a weighted, directed graph with weight function w:E→R, and let (u,v)∈E.
- Then, immediately after relaxing edge (u,v) by executing RELAX(u,v,w), we have v.d ≤ u.d+w(u,v).
- Proof: if, just prior to relaxing edge (u,v), we have v.d>u.d+w(u,v), then before v.d=u.d+w(u,v) afterward. If, instead, v.d≤u.d+w(u,v) just before the relaxation, then neither u.d nor v.d changes, and so v.d≤u.d+w(u,v) afterward.

Convergence Property (Lemma 24.14)

- Let $G=(V,E)$ be a weighted, directed graph with weight function $w : E \rightarrow R$.
- Let $s \in V$ the source vertex.
- Let $s \rightsquigarrow u \rightarrow v$ be a shortest path in G for some vertices $u,v \in V$.
- Let the graph is initialized by INITILIZE-SINGLE-SOURCE(G,s) and then a sequence of relaxation steps that includes the call RELAX(u,v,w) is executed on the edge of G.
- If $u.d = \delta(s,u)$ at any time prior to the call, then we have $v.d = \delta(s,v)$ at all times after the call.

Convergence Property (Proof)

- By the upper-bound property, if $u.d = \delta(s,u)$ at some point prior to relaxing edge (u,v), this equality holds thereafter.
- In particular, after relaxing edge (u,v), we have $v.d \leq u.d+w(u,v) = \delta(s,u)+w(u,v) = \delta(s,v)$.
- By the upper-bound property, $v.d \geq \delta(s,v)$ from which we conclude that $v.d = \delta(s,v)$.
- Thus, this equality is maintained thereafter.

Shortest-paths Properties

- Triangle Inequality
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u)+w(u,v)$.
- Upper-bound property
  - We always have $v.d \geq \delta(s,v)$ for all $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$, it never changes.
- No-path property
  - If there is no path from s to v, then we always have $v.d = \infty = \delta(s,v)$.
- Convergence property
  - If $s \rightsquigarrow u \rightarrow v$ be a shortest path in G for some vertices $u,v \in V$, 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)

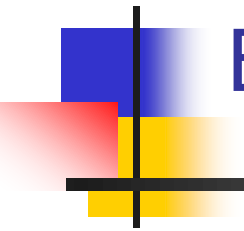
- If $p=\langle v_0,v_1,\ldots,v_k\rangle$ is a shortest path from $s=v_0$ to v_k, and the edges of p are relaxed in the order $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$, then $v_k.d=\delta(s,v_k)$.
- This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Path-relaxation Property (Proof)

- We show by induction that after i-th edge of path $p=\langle v_0,v_1,\ldots,v_k\rangle$ is relaxed, we have $v_i.d=\delta(s,v_i)$.
- For the basis, i=0, and before any edges of p have been relaxed, we have from the initialization that $v_0.d=s.d=0=\delta(s,s)$. By the upper-bound property, the value of s.d never changes after initialization.
- For the induction step, we assume that $v_{i-1}.d = \delta(s,v_{i-1})$, and we examine the relaxation of edge (v_{i-1},v_i). By the convergence property, after relaxation, we have $v_i.d=\delta(s,v_i)$, and this equality is maintained at all times thereafter.

Predecessor-subgraph Property (Lemma 24.17)

- Let $G=(V,E)$ be a weighted, directed graph with weight function $w:E \rightarrow R$, let $s \in V$ be a source vertex, and assume that G contains no negative-weight cycles that are reachable from s.
- Let us call INITIALIZE-SINGLE-SOURCE(G, s) and then execute any sequence of relaxation steps on edges of G that produces $v.d=\delta(s,v)$ for all $v \in V$.
- Then, the predecessor subgraph G is a shortest-path tree rooted at s.
- Proof is omitted.



Bellman-Ford Algorithm

Bellman-Ford Algorithm

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

Bellman-Ford Algorithm

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

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

Test whether negative -weight cycle exists

- **Edge weights may be negative**

Bellman-Ford Algorithm

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

- **Running time O(|V||E|)**

Lemma 24.2

- Let G=(V,E) be a weighted, directed graph with source s and weight function w:E→R.
- Assume that G contains no negative-weight cycles that are reachable from s.
- Then, after the |V|-1 iterations of the **for** loop of lines 2 - 4 of BELLMAN-FORD, we have v.d=δ(s,v) for all vertices that are reachable from s.

Lemma 24.2 (Proof)

- We prove the lemma by appealing to the path-relaxation property.
- Consider any vertex that is reachable from s, and let $p=\langle v_0,v_1,\ldots,v_k\rangle$, where $v_0= s$ and $v_k = v$, be any shortest path from s to v.
- Because shortest paths are simple, p has at most |V|-1 edges, and so $k \leq |V|-1$. Each of the |V|-1 iterations of the **for** loop of lines 2–4 relaxes all |E| edges.
- The edge (v_{i-1}, v_i) is one among the edges relaxed in the i-th iteration, for i = 1, 2, …, k.
- By the path-relaxation property, therefore, $v.d=v_k.d=\delta(s,v_k)=\delta(s,v)$.

Corollary 24.3

- Let G=(V,E) be a weighted, directed graph with source vertex s and weight function w:E→R.
- Assume that G contains no negative-weight cycles that are reachable from s.
- Then, for each vertex v ∈ V, there is a path from s to v if and only if BELLMAN-FORD terminates with v.d < ∞ when it is run on G.
- The proof is left as Exercise 24.1-2.

Correctness of the Bellman-Ford Algorithm

- Let BELLMAN-FORD be run on a weighted, directed graph $G=(V,E)$ with source s and weight function $w:E \rightarrow R$.
- If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d=\delta(s,v)$ for all vertices $v \in V$, and the predecessor subgraph G is a shortest-paths tree rooted at s.
- It G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE

Correctness of the Bellman-Ford Algorithm (Proof)

- Suppose that G contains no negative-weight cycles that are reachable from s.
- We first prove the claim that at termination, $v.d=\delta(s,v)$ for all vertices $v \in V$.
  - If vertex v is reachable from s, then Lemma 24.2 proves this.
  - If v is not reachable from s, then the claim follows from the no-path property.
  - Thus, the claim is proven.
- The predecessor-subgraph property, along with the claim, implies that $G.\pi$ is a shortest-paths tree.
- Now we use the claim to show that BELLMAN-FORD returns TRUE.
- At termination, we have for all edges $(u,v) \in E$,
  - $v.d = \delta(s,v)$
    $\leq \delta(s,u) + w(u,v)$ (by the triangle inequality)
    $= u.d+w(u,v)$
- and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE.
- Thus, it returns TRUE.

Correctness of the Bellman-Ford Algorithm (Proof)

- Suppose G contains a negative-weight cycle that is reachable from s.
- Let this cycle c=<v_0,v_1,…,v_k>, where $v_0=v_k$ and $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ **(24.1)**
- Assume Bellman-Fold algorithm returns TRUE.

  $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1,2,\dots,k$
- Summing the inequalities around cycle c

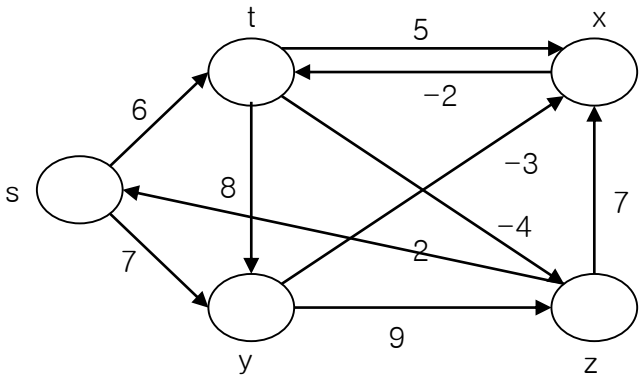
$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} \left(v_{i-1}.d + w(v_{i-1}, v_i)\right)$$

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

- Since $v_0=v_k$, each vertex in c appears exactly once in each of the summations, $\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$.
- Moreover, by Corollary 24.3, v_i.d is finite for i=1,2,…,k.
- Thus, $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$ which contradicts inequality **(24.1)**

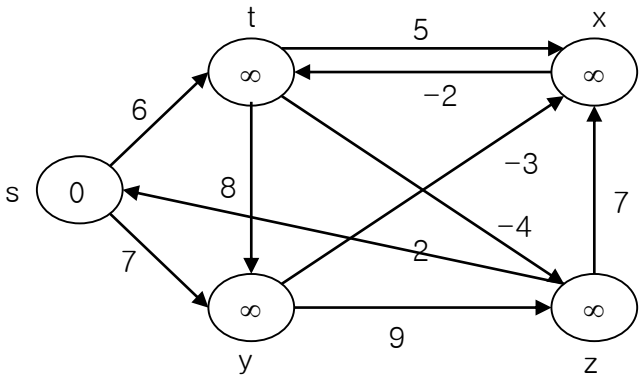
Bellman-Ford Algorithm

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



Bellman-Ford Algorithm

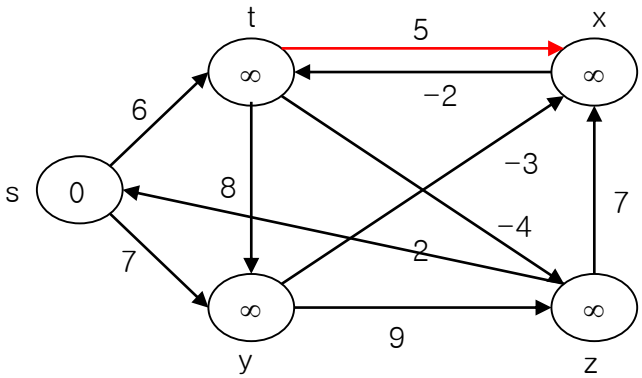
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



Bellman-Ford Algorithm

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

i = 1

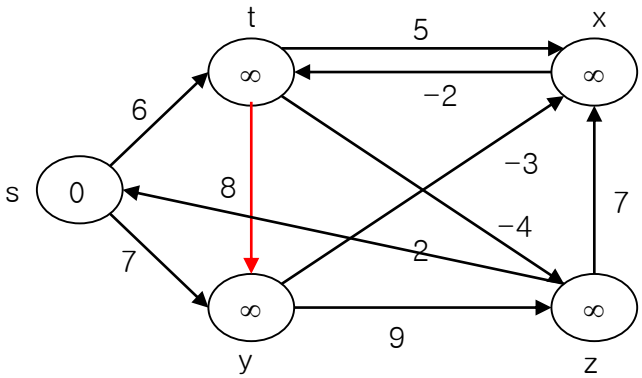


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

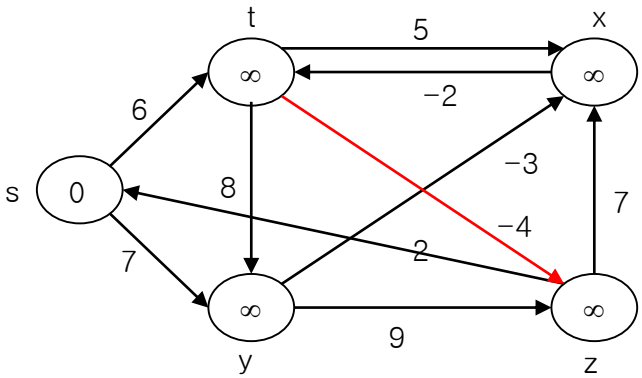


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

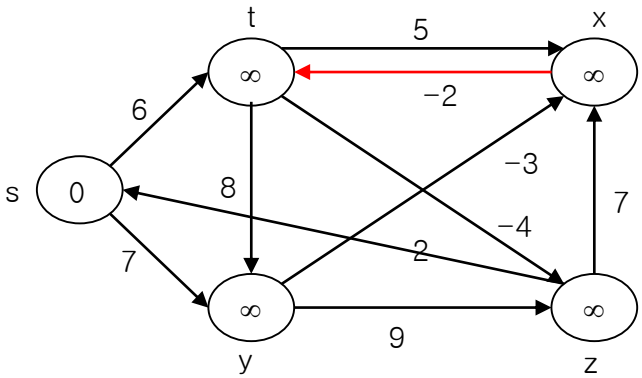


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

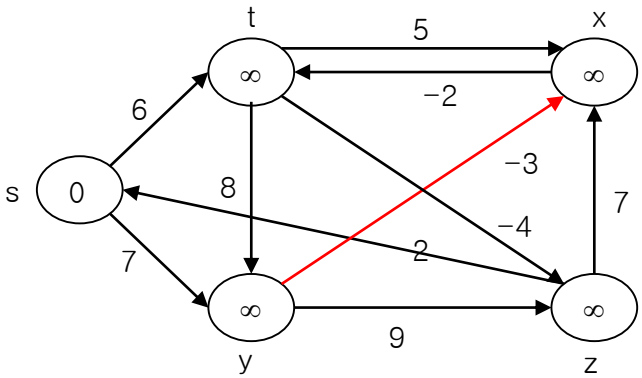


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

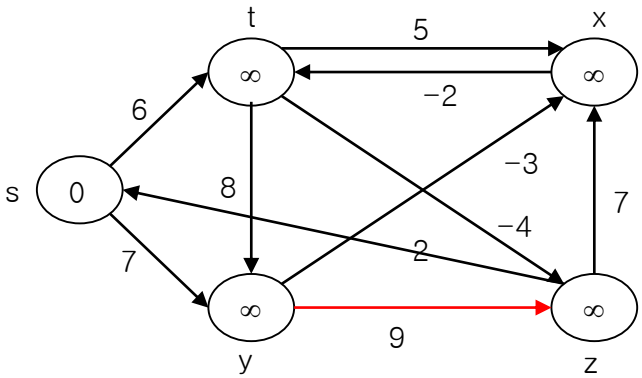


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

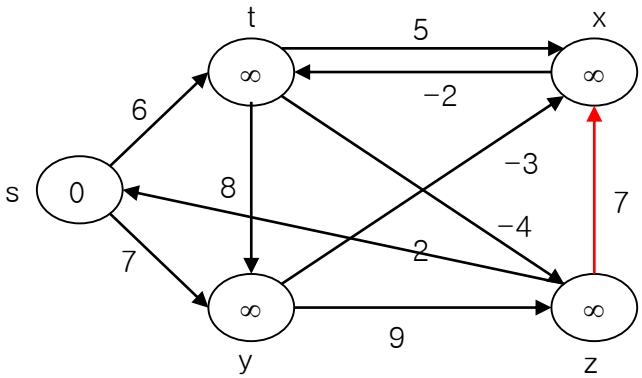


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

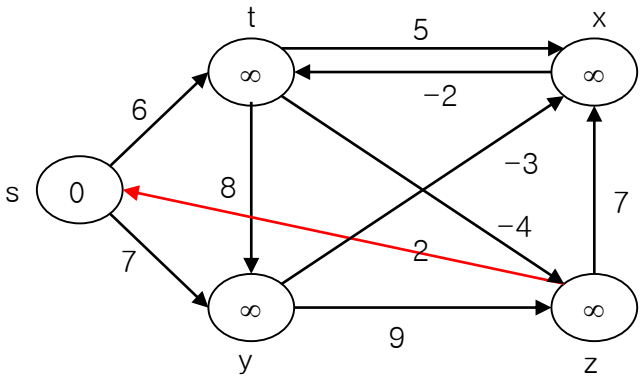


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

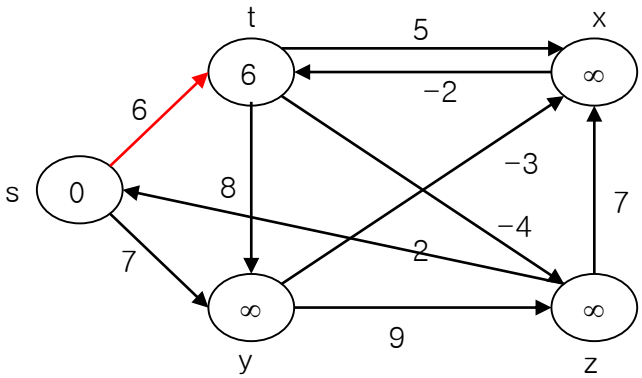


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

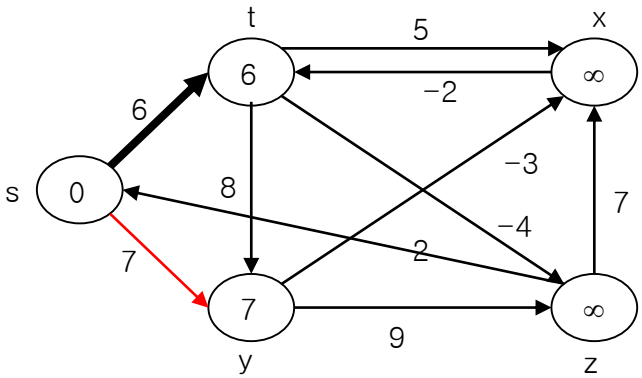


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 1

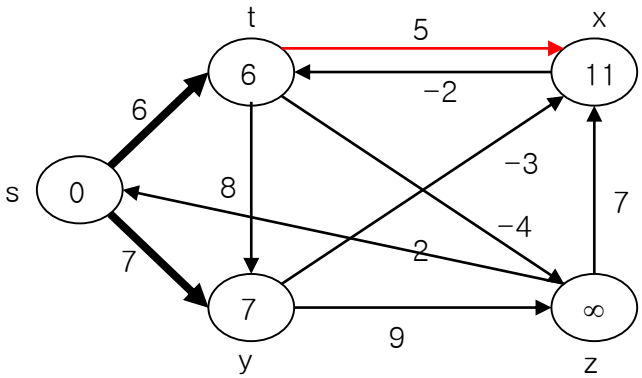


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

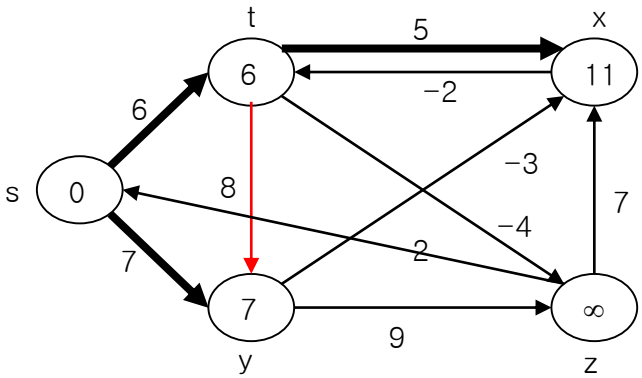


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

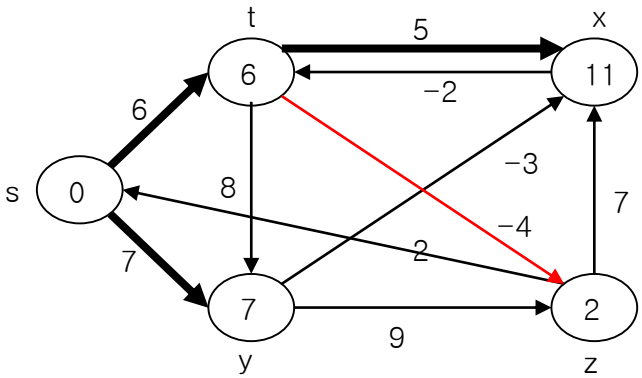


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

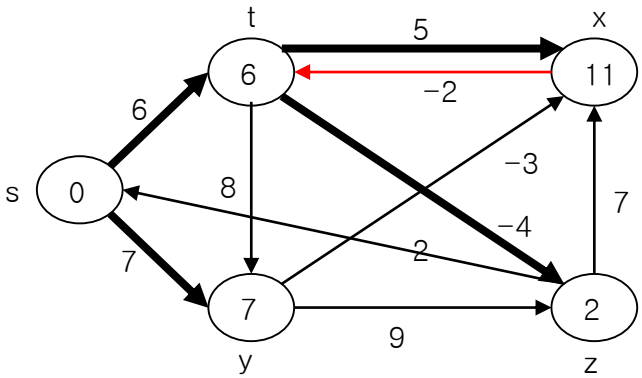


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

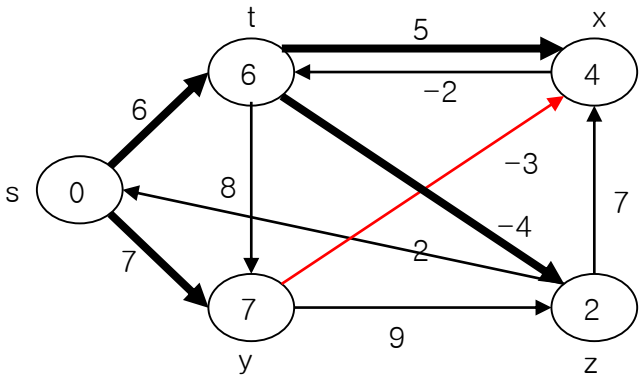


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

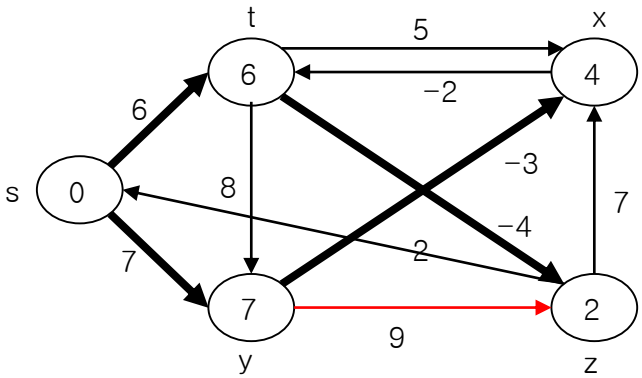


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

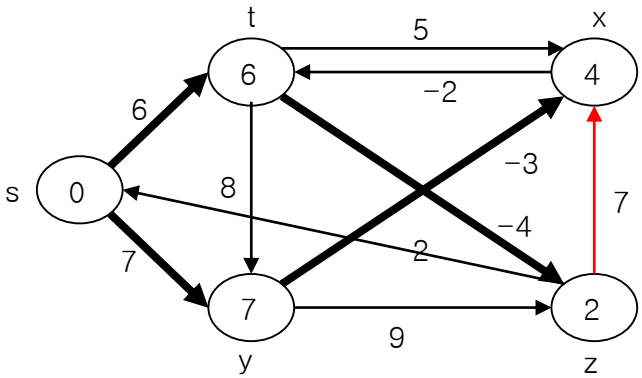


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

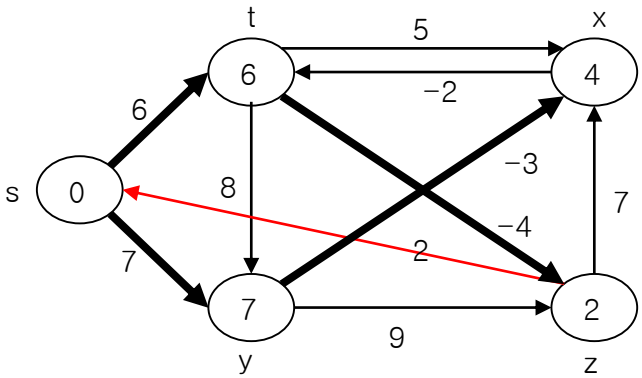


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

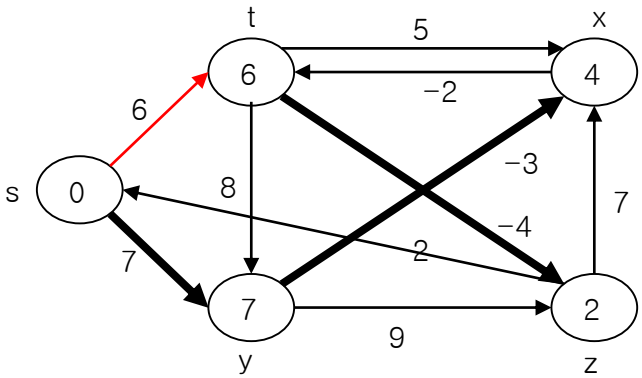


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

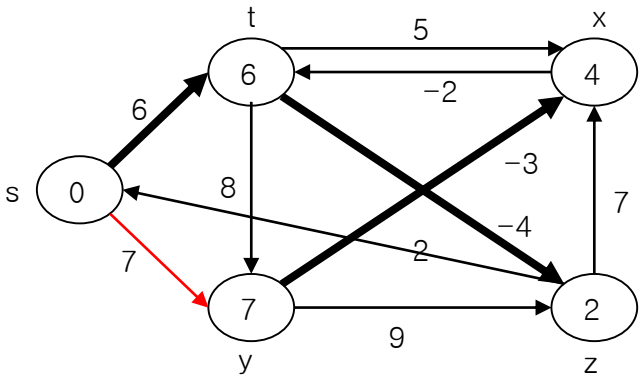


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 2

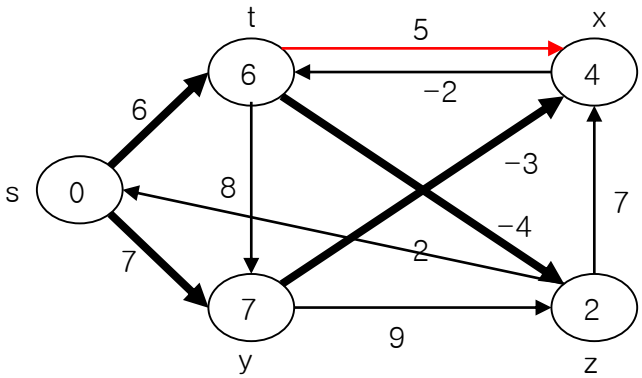


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

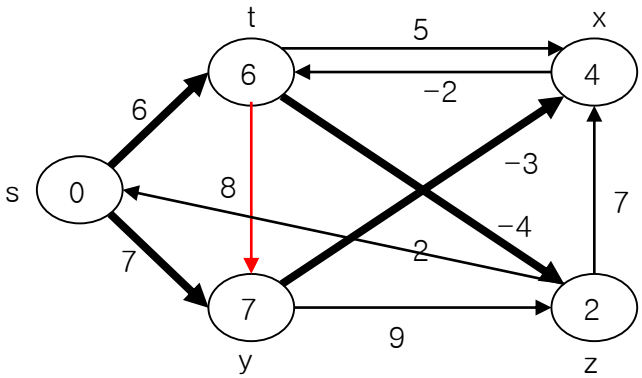


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

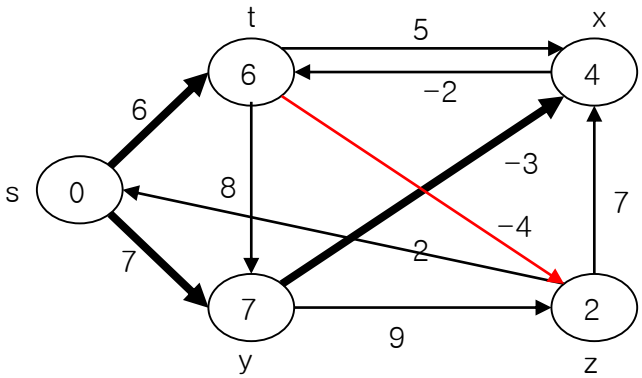


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

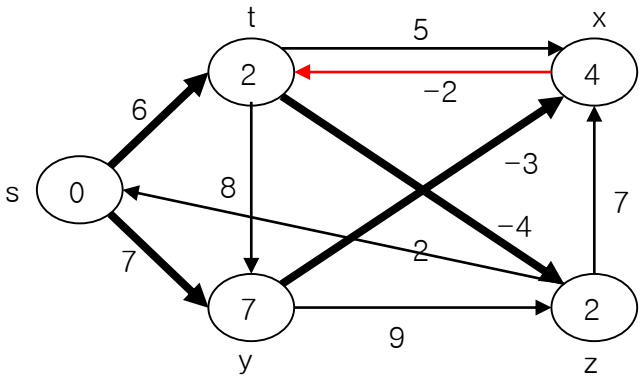


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

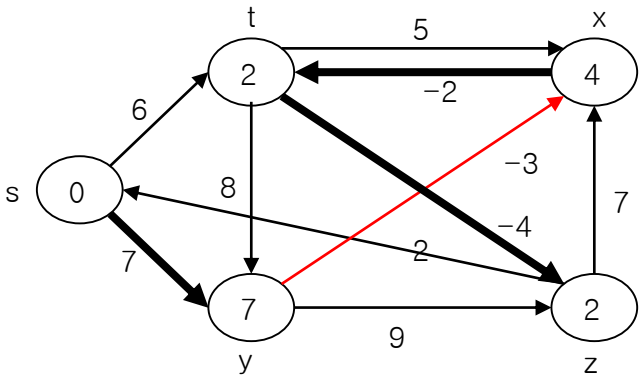


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

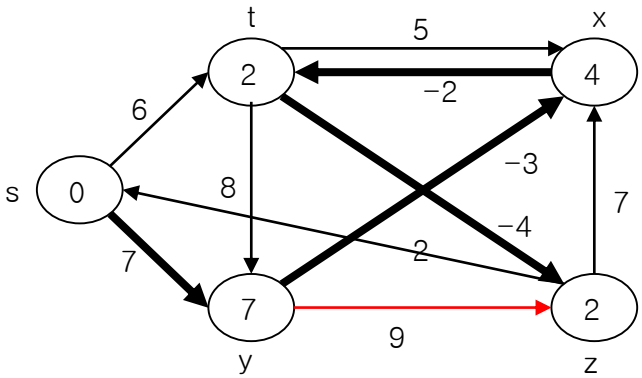


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

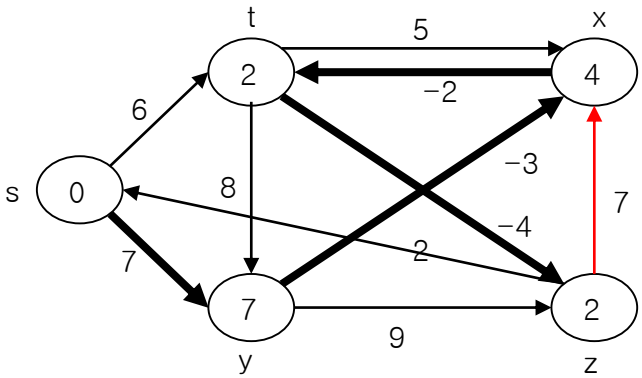


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

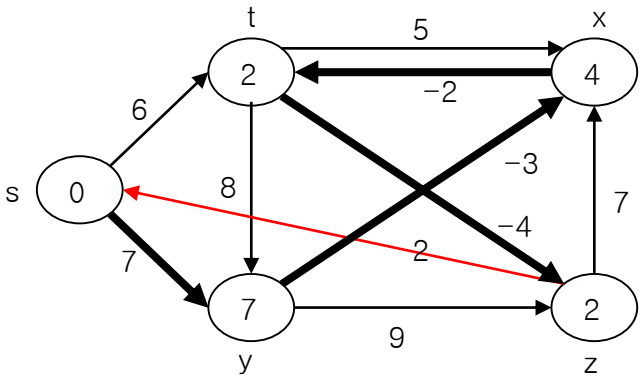


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

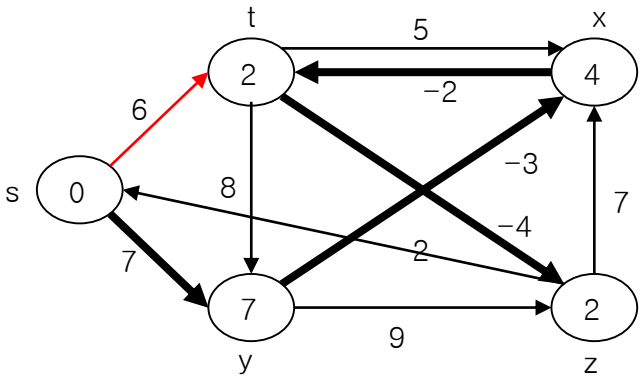


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

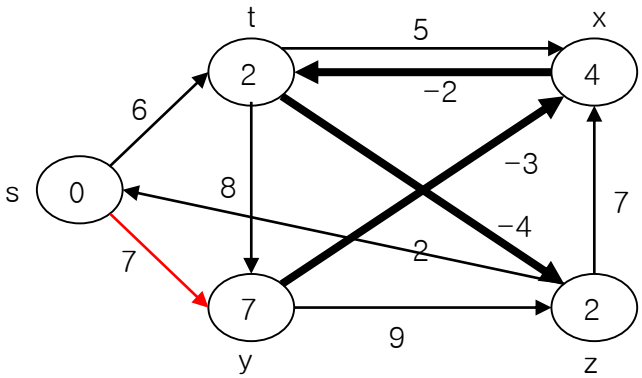


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 3

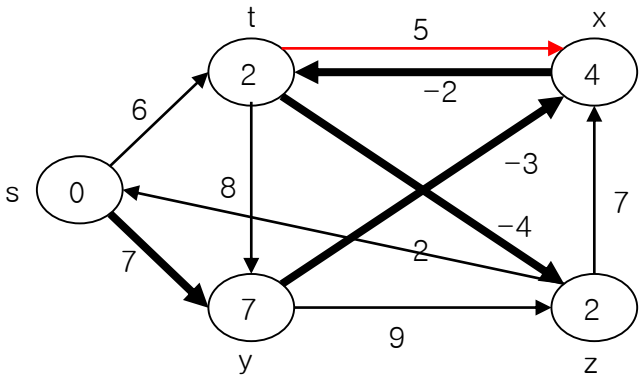


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

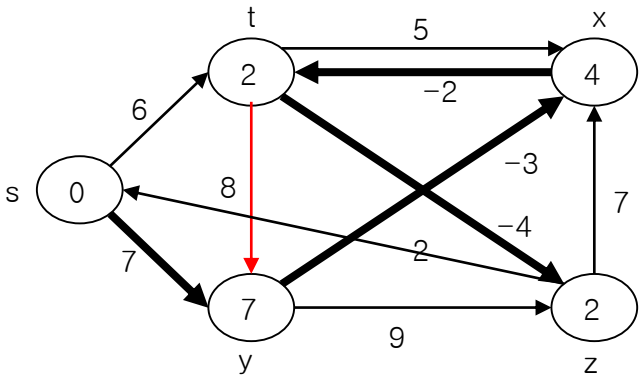


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

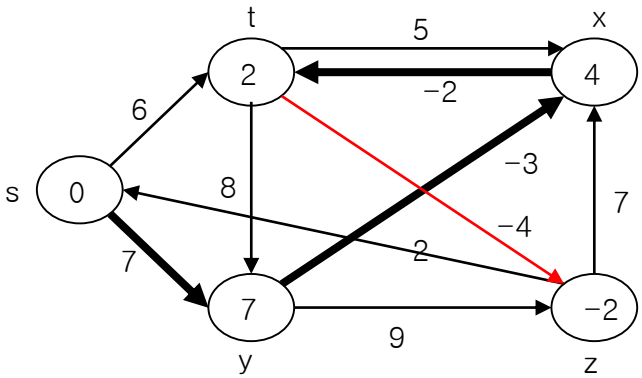


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

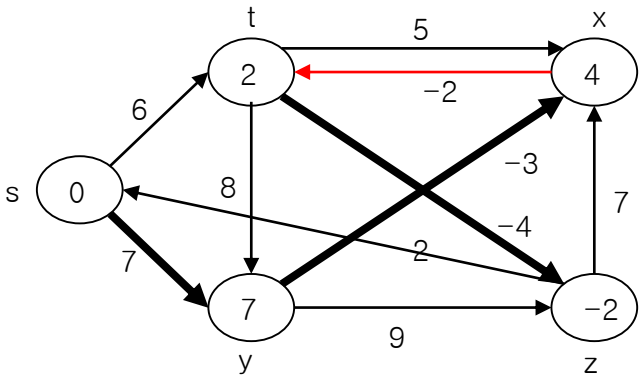


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

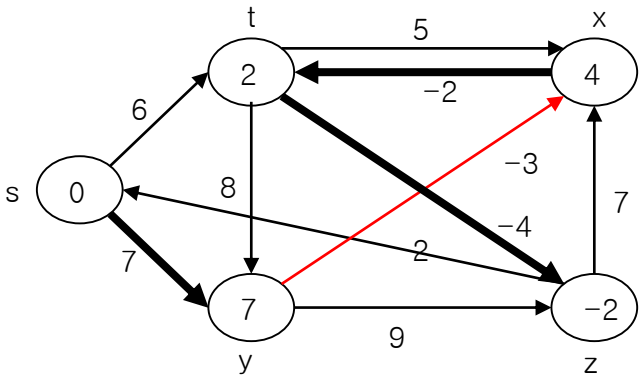


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

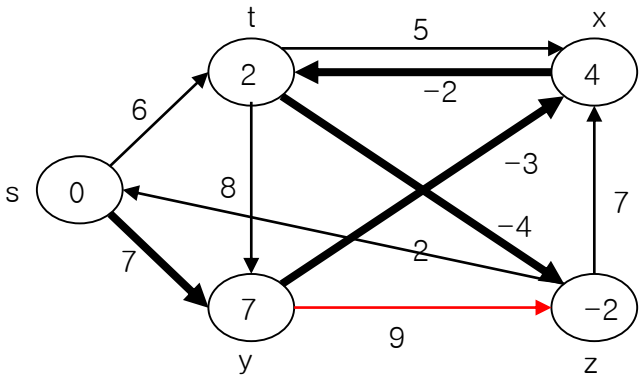


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

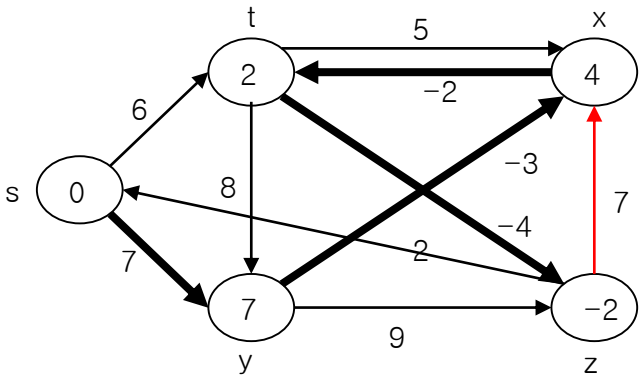


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

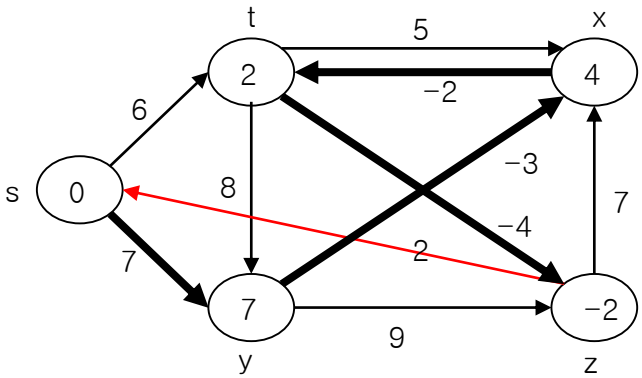


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

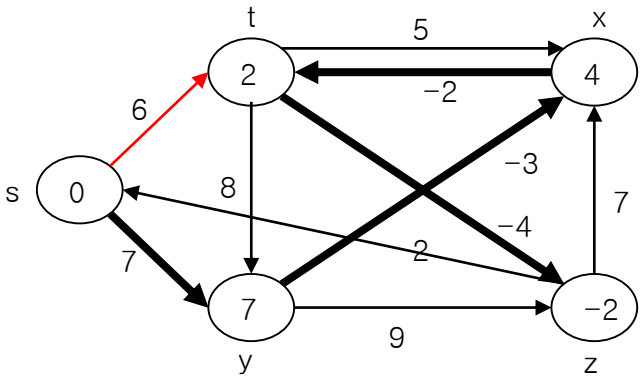


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

i = 4

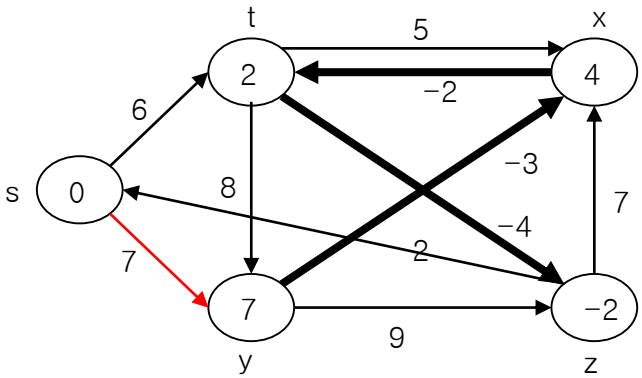


G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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

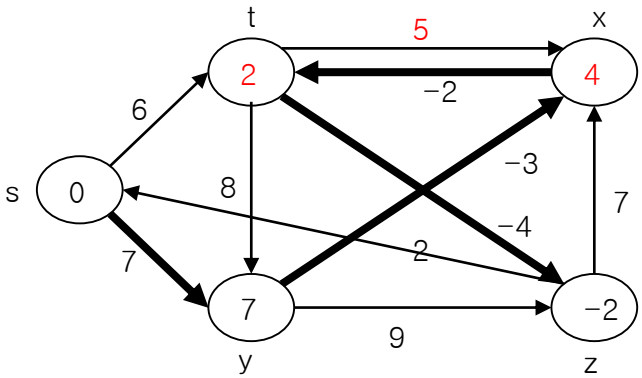
i = 4



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

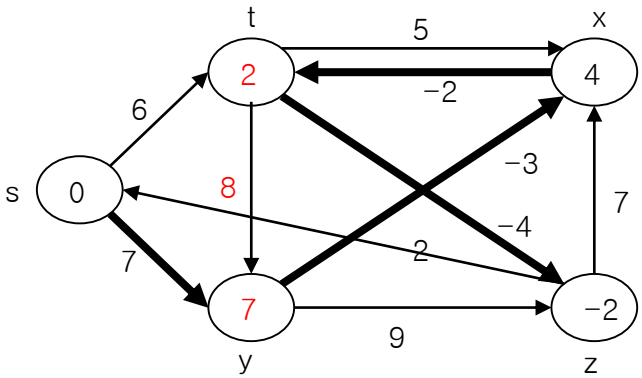
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

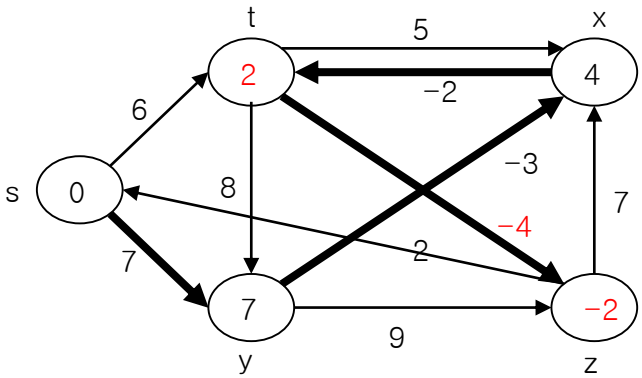
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

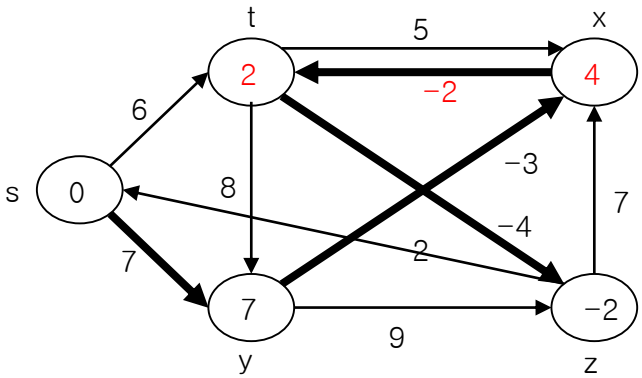
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

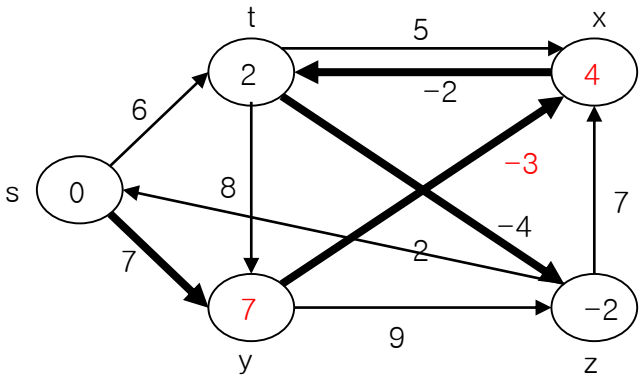
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.         for each edge (u,v) ∈ G.E
4.            RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.         if  v.d > u.d + w(u,v)
7.            return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

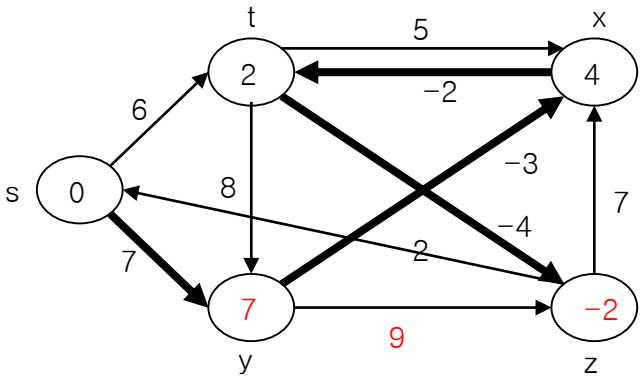
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

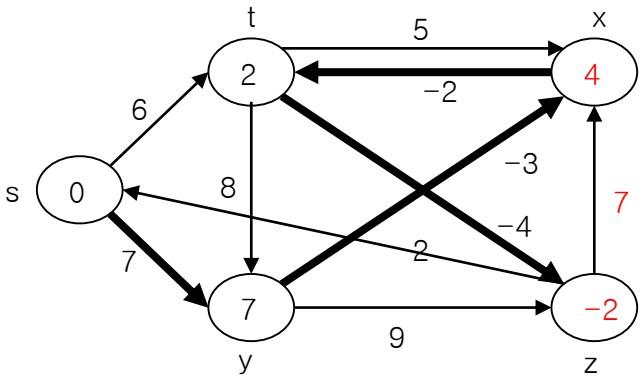
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

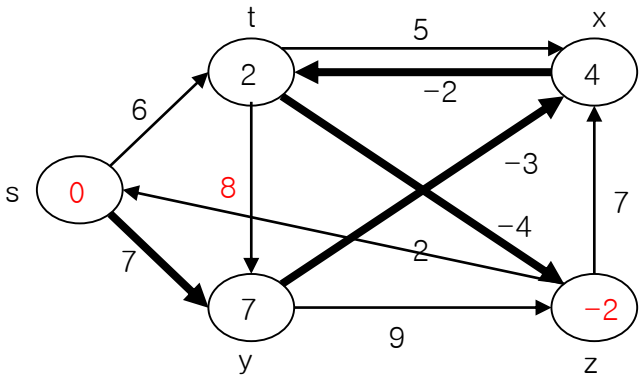
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.         for each edge (u,v) ∈ G.E
4.            RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.         if  v.d > u.d + w(u,v)
7.            return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

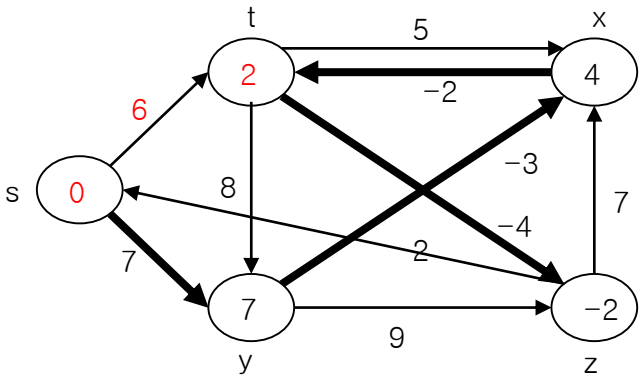
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.         for each edge (u,v) ∈ G.E
4.            RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.         if  v.d > u.d + w(u,v)
7.            return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

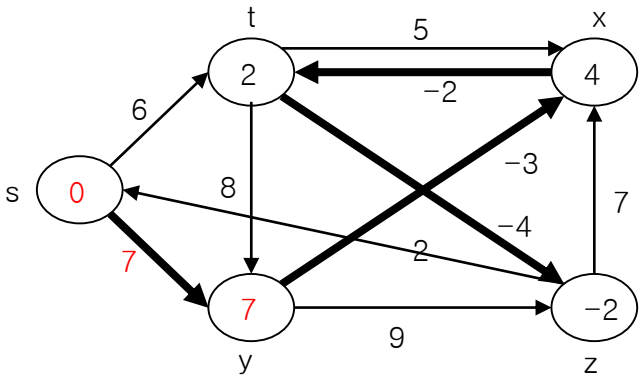
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

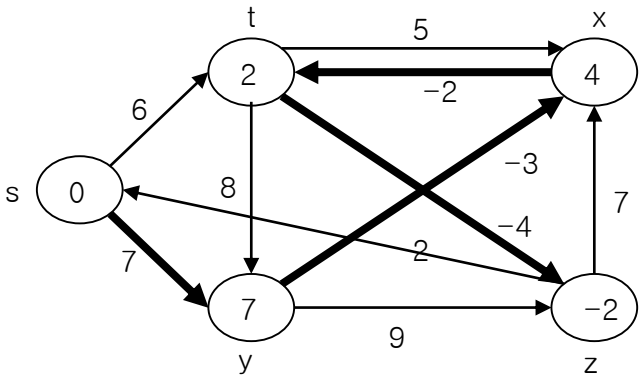
```
BELLMAN-FORD(G, w, s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      for i=1 to |G.V|-1
3.          for each edge (u,v) ∈ G.E
4.              RELAX(u, v, w)
5.      for each edge (u,v) ∈ G.E
6.          if  v.d > u.d + w(u,v)
7.              return FALSE
8.      return TRUE
```



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Bellman-Ford Algorithm

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



G.E	(t,x)	(t,y)	(t,z)	(x,t)	(y,x)	(y,z)	(z,x)	(z,s)	(s,t)	(s,y)

Single-source Shortest Paths in Directed Acyclic Graphs (DAG)

Single-source Shortest Paths in Directed Acyclic Graphs

- Problem: finding shortest paths in DAG
  - Bellman-Ford takes O(|V||E|) time.
  - How can we do better?
  - Idea: use the topological sort.

DAG Shortest Path

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

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

DAG shortest path

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

1. topologically sort the vertices of G ← O(|V|+|E|)
2. INITIALIZE-SINGLE-SOURCE(G,s) ← O(|V|)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

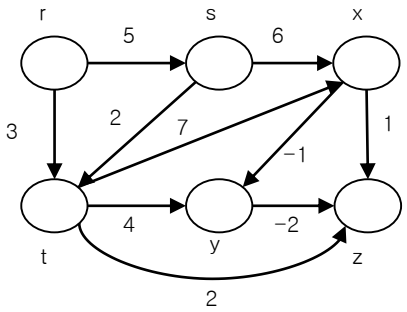
O(|E|)

running time is O(|V|+|E|)

DAG Shortest Path

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

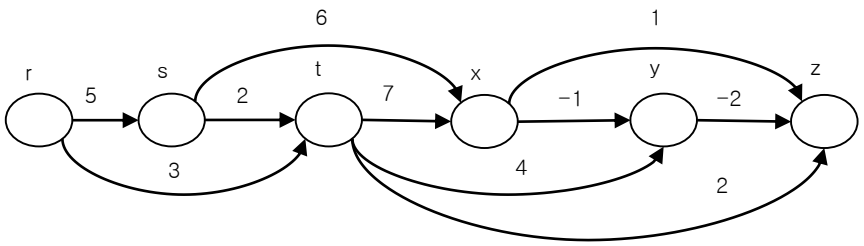
1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)



DAG Shortest Path

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

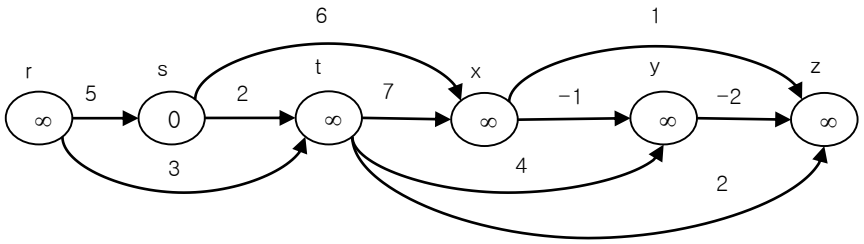
1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)



DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

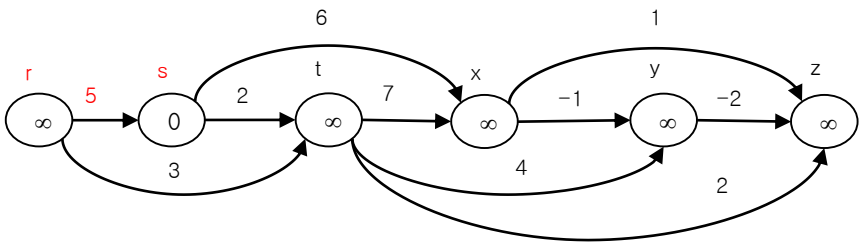


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	s	t				

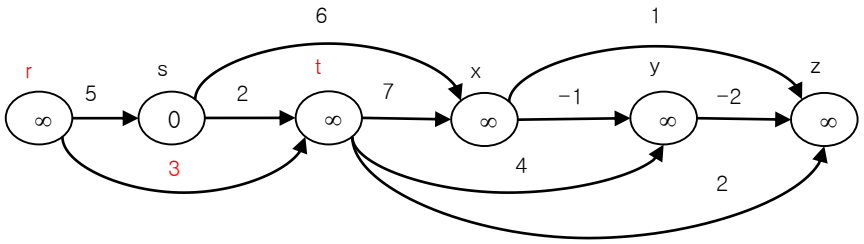


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	s	t				

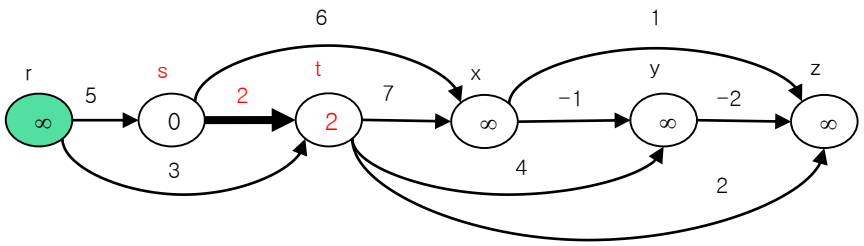


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	t	x				

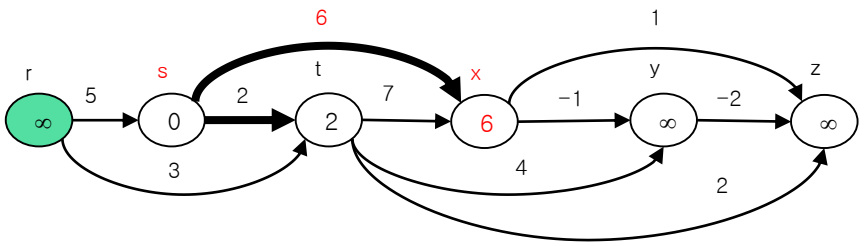


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	t	x				

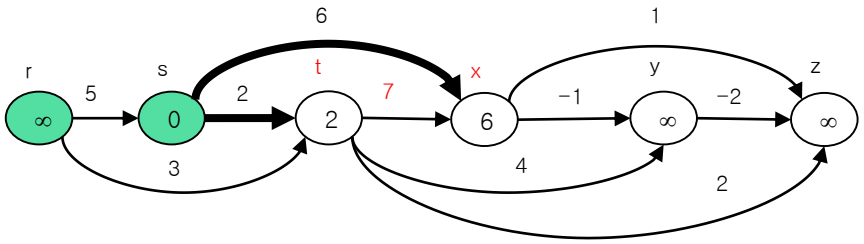


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	x	y	z			

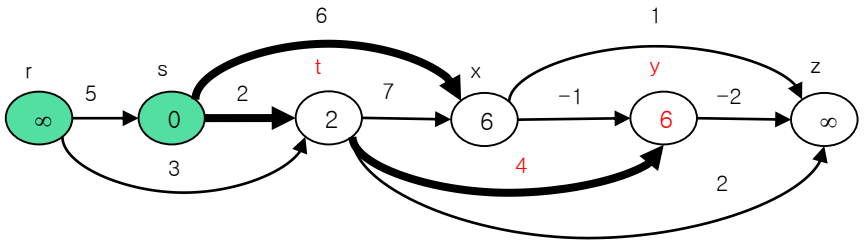


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	x	y	z			

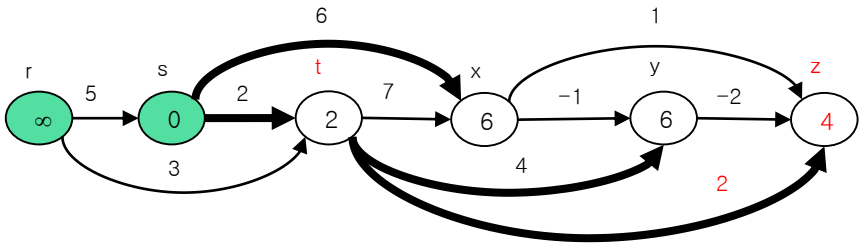


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	x	y	z			



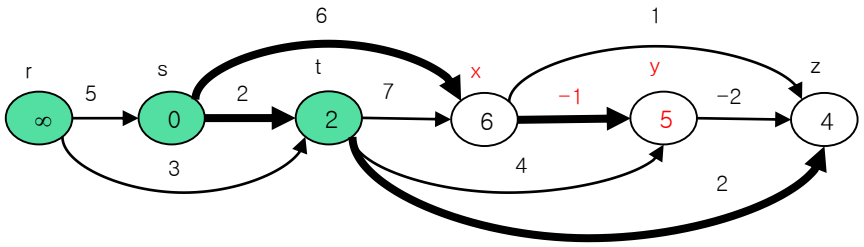
DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z

G.Adj[u]	y	z

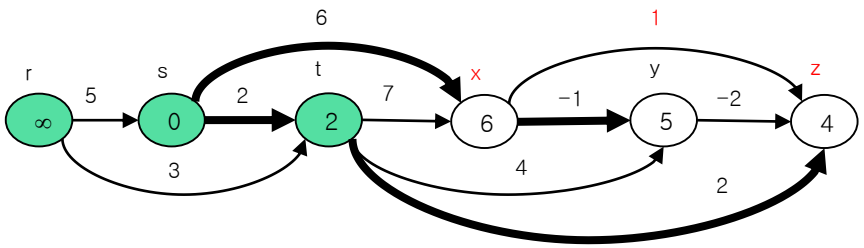


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	y	z				

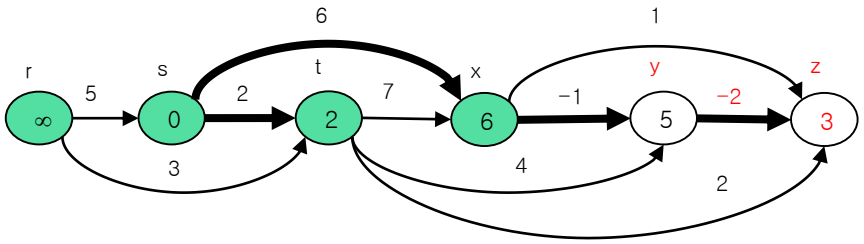


DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u	r	s	t	x	y	z
G.Adj[u]	z					



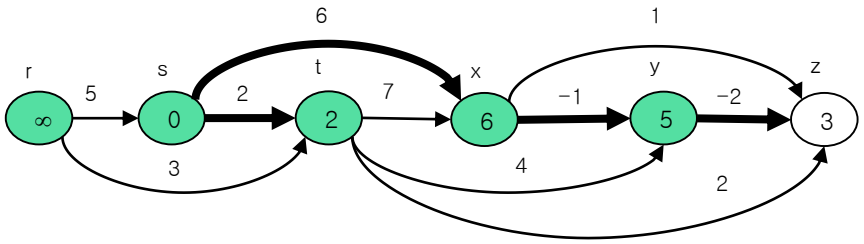
DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)

u

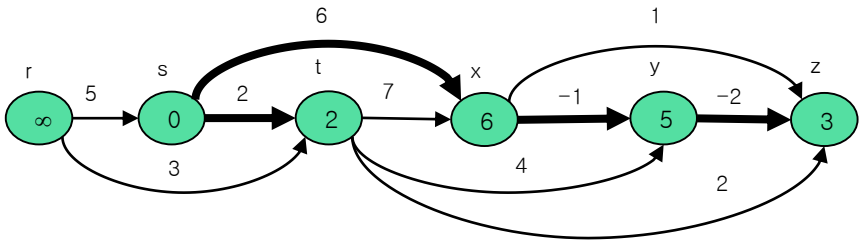
r	s	t	x	y	z



DAG Shortest Path

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

1. topologically sort the vertices of G
2. INITIALIZE-SINGLE-SOURCE(G,s)
3. **for** each vertex u, taken in topologically sorted order
4. **for** each vertex v∈ G.Adj[u]
5. RELAX(u,v,w)



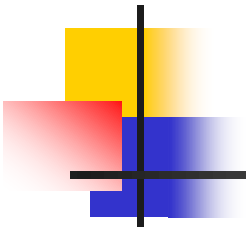
Theorem 24.5

- If a weighted, directed graph G=(V,E) has source vertex s and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, v.d=δ(s,v) for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.

Theorem 24.5 (Proof)

- We first show that $v.d=\delta(s,v)$ for all vertices $v \in V$ at termination.
- If v is not reachable from s, then $v.d=\delta(s,v)=\infty$ by the no-path property.
- Now, suppose that v is reachable from s, so that there is a shortest path $p=\langle v_0,v_1,\ldots,v_k\rangle$ where $v_0 = s$ and $v_k=v$.
- Because we process the vertices in topologically sorted order, we relax the edges on p in the order $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$.
- The path-relaxation property implies that $v.d=\delta(s,v)$ at termination for $i = 1, 2, \ldots, k$.
- Finally, by the predecessor subgraph property, G_π is a shortest-paths tree.

Dijkstra’s Algorithm

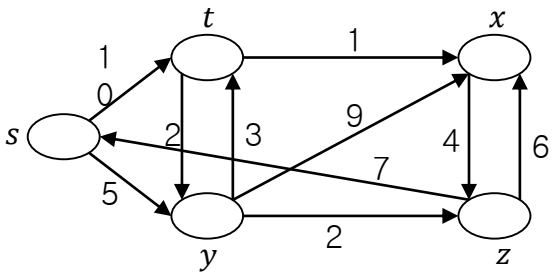


Dijkstra's Algorithm

- On weighted, directed graph G=(V,E) for which all edge weights are nonnegative.
- The running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

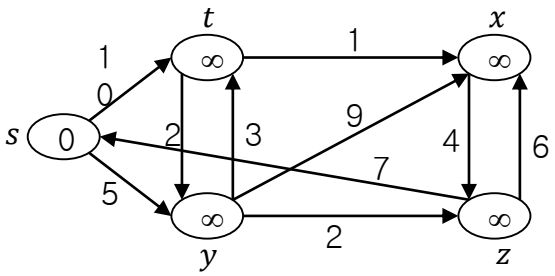
Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```



Dijkstra's Algorithm

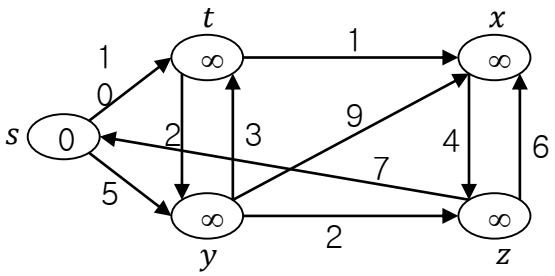
```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

S=∅



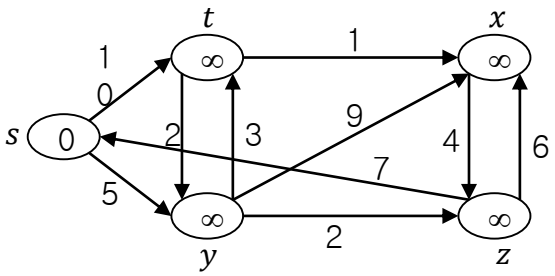
Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

S=∅

Q

G.V	*s*	*t*	*x*	*y*	*z*
d	0	∞	∞	∞	∞



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

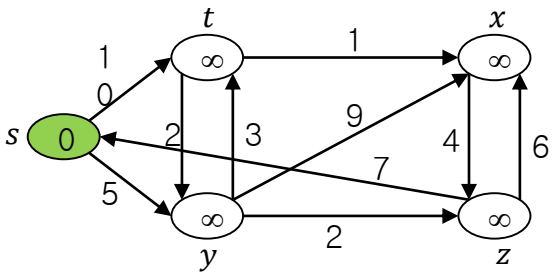
$S=\{s\}$

Q

G.V	t	x	y	z
d	∞	∞	∞	∞

$u=s$

$G.adj[s] = \{t, y\}$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

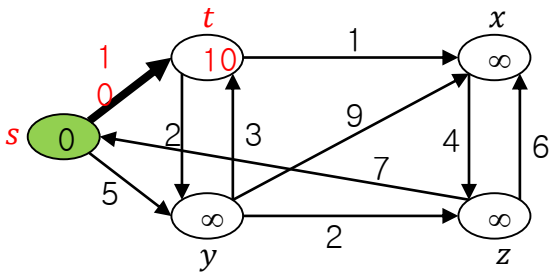
$S=\{s\}$

Q

G.V	t	x	y	z
d	10	∞	∞	∞

$u=s$

$G.adj[s] = \{t, y\}$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

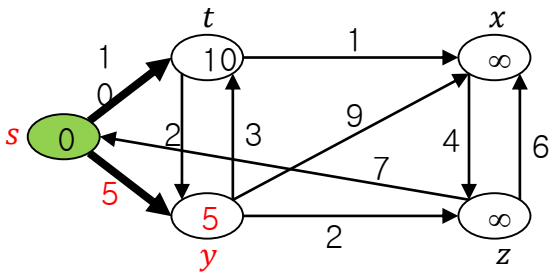
$S=\{s\}$

Q

G.V	t	x	y	z
d	10	∞	5	∞

$u=s$

$G.adj[s] = \{t, y\}$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

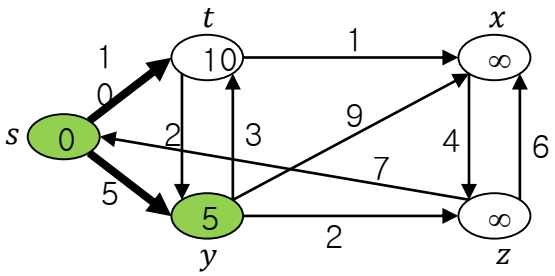
S={s, y}

Q

G.V	t	x	z
d	10	∞	∞

u=y

G.adj[y] = {t, x, z}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

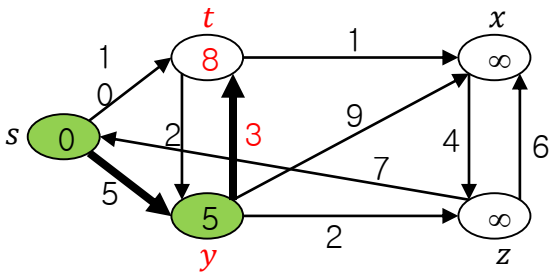
S={s, y}

Q

G.V	t	x	z
d	8	∞	∞

u=y

G.adj[y] = {t, x, z}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

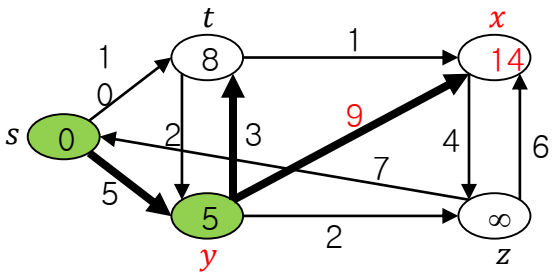
S={s, y}

Q

G.V	t	x	z
d	8	14	∞

u=y

G.adj[y] = {t, x, z}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

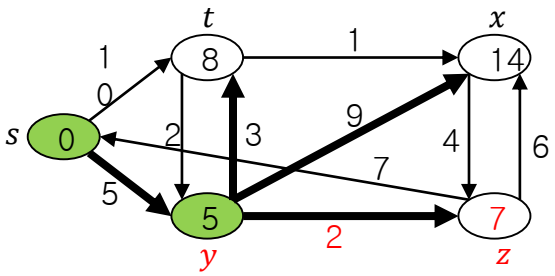
S={s, y}

Q

G.V	t	x	z
d	8	14	7

u=y

G.adj[y] = {t, x, z}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

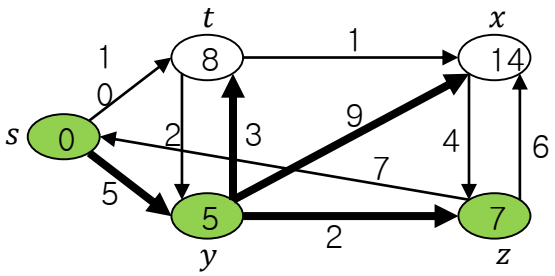
$S=\{s, y, z\}$

Q

G.V	t	x
d	8	14

$u=z$

$G.adj[z] = \{x, s\}$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

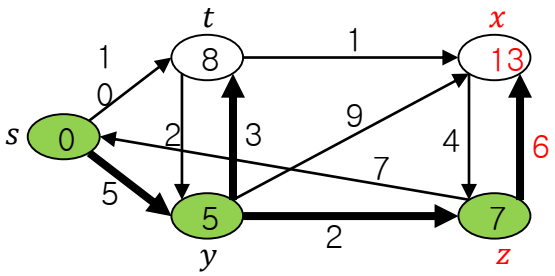
S={s, y, z}

Q

G.V	t	x
d	8	13

u=z

G.adj[z] = {x, s}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

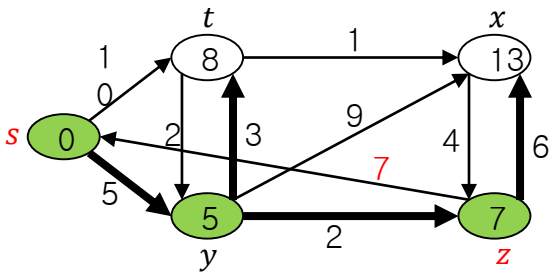
$S=\{s, y, z\}$

Q

G.V	t	x
d	8	13

$u=z$

$G.adj[z] = \{x, s\}$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

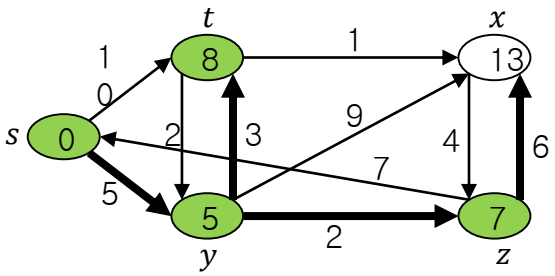
S={s, y, z, t}

Q

G.V	x
d	13

u=t

G.adj[t] = {x, y}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

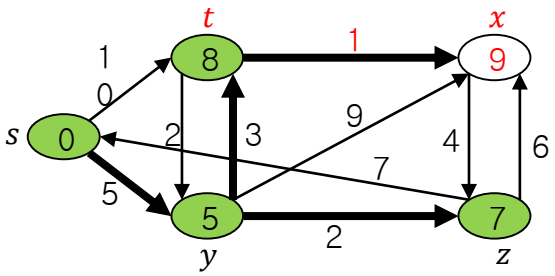
S={s, y, z, t}

Q

G.V	x
d	9

u=t

G.adj[t] = {x, y}



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

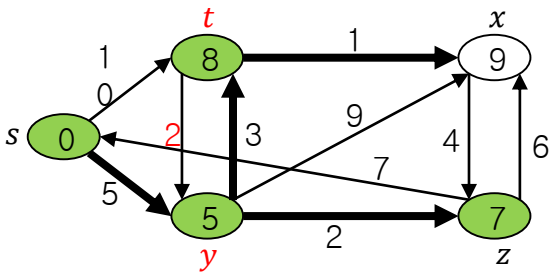
S=$\{s, y, z, t\}$

Q

G.V	x
d	9

u=t

G.adj$[t]$ = $\{x, y\}$



Dijkstra's Algorithm

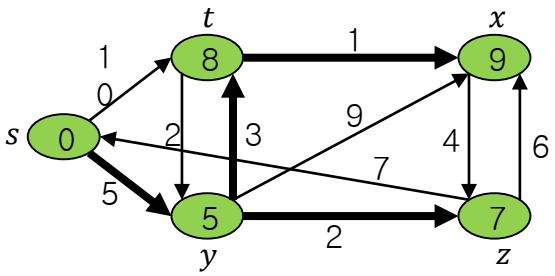
```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

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

$Q=\emptyset$

$u=x$

$G.adj[x] = \{z\}$



Dijkstra's Algorithm

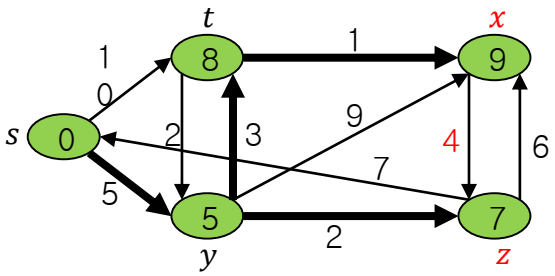
```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

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

$Q=\emptyset$

$u=x$

$G.adj[x] = \{z\}$

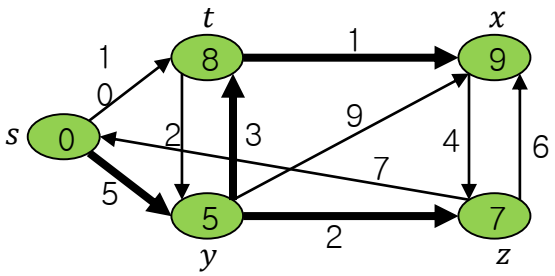


Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

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

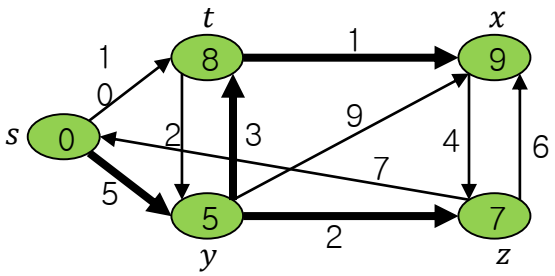
$Q=\emptyset$



Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

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



Running Time of Dijkstra's Algorithm

- It depends on implementations of the min-priority queue Q.
- If we implement Q as a binary min-heap,
  - EXTRACT-MIN takes O(lg |V|) time.
  - DECREASE-KEY takes O(lg |V|) time.
- If we implement Q as a simple array,
  - EXTRACT-MIN takes O(|V|) time.
  - DECREASE-KEY O(1) time.
- If we implement Q as a Fibonacci heap,
  - EXTRACT-MIN takes O(lg |V|) amortized time.
  - DECREASE-KEY O(1) amortized time.

Dijkstra's Algorithm

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)
2.      S = ∅
3.      Q = G.V
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)
6.          S = S U {u}
7.          for each vertex v ∈ G.Adj[u]
8.              RELAX(u,v,w)
```

Dijkstra's Algorithm

- min-priority queue : array

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)      ← O(|V|)
2.      S = ∅                              ← O(1)
3.      Q = G.V                            ← O(|V|)
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)             ← O(|V|²)
6.          S = S U {u}                    ← O(|V|)
7.          for each vertex v ∈ G.Adj[u]   ← O(|E|)
8.              RELAX(u,v,w)
```

Dijkstra's algorithm running time is $O(|V|^2)$

Dijkstra's Algorithm

- min-priority queue : binary min-heap

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)   ← O(|V|)
2.      S = ∅                           ← O(1)
3.      Q = G.V                         ← O(|V|)
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)          ← O(|V| lg |V|)
6.          S = S U {u}                 ← O(|V|)
7.          for each vertex v ∈ G.Adj[u] ← O(|E| lg |V|)
8.              RELAX(u,v,w)
```

- Running time:
  - O((|V|+|E|) lg |V|) , if all vertices are reachable →O(|E| lg |V|).
  - Better than $O(|V|^2)$, if the graph is sufficiently sparse: $|E| = o(|V|^2 / \lg |V|)$.

Dijkstra's Algorithm

- min-priority queue : Fibonacci heap

```
DIJKSTRA(G,w,s)
1.      INITIALIZE-SINGLE-SOURCE(G,s)      ← O(|V|)
2.      S = ∅                              ← O(1)
3.      Q = G.V                            ← O(|V|)
4.      while Q ≠ ∅
5.          u = Extract-Min(Q)             ← O(|V| lg |V|)
6.          S = S U {u}                    ← O(|V|)
7.          for each vertex v ∈ G.Adj[u]   ← O(|E|)
8.              RELAX(u,v,w)
```

Dijkstra's algorithm running time is O(|V| lg |V|+ |E|)

Theorem 24.6 (Correctness of Dijkstra's Algorithm)

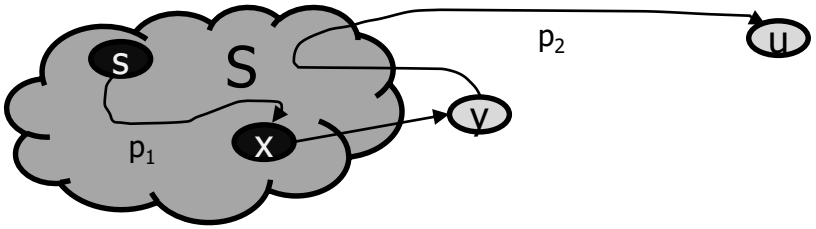
- Dijkstra's algorithm, run on a weighted, directed graph $G=(V,E)$ with non-negative weight function w and source s, terminates with $u.d=\delta(s,u)$ for all vertices $u \in V$.

Theorem 24.6 (Proof)

- Loop invariant
- At the start of each iteration of the while loop of lines 4-8, $v.d = \delta(s,v)$ for each vertex v in S
- Initialization: S={}, so true
- Maintenance:
  - Let u be the first vertex for which $u.d \neq \delta(s,u)$ when it is added to set S
  - $u \neq s$ because s is the first vertex added to set S and $s.d = \delta(s,s)=0$
  - Because $u \neq s$, we also have that $S \neq \{\}$ just before u is added to S
  - There must be some path from s to u, for otherwise $u.d=\delta(s,u)= \infty$ by no-path property
  - There is a shortest path p from s to u
  - Prior to adding u to S, path p connects a vertex in S, namely s, to a vertex in V-S, namely u.

Theorem 24.6 (Proof)

- Let us consider the first vertex y along p such that $y \in V\text{-}S$, and let $x \in S$ be y's predecessor along p
- Figure shown below illustrates, we can decompose path p into $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$ ($s \rightsquigarrow x$: p_1 , $y \rightsquigarrow u$: p_2)
- Claim: $y.d = \delta(s,y)$ when u is added to S
  - $x.d = \delta(s,x)$ when x was added to s
    ($\because$ we chose u as the first vertex for which $u.d \neq \delta(s,u)$)
  - Edge (x,y) was relaxed at that time, and the claim follows from the convergence property



Theorem 24.6 (Proof)

- We can now obtain a contradiction to prove u.d= δ(s,u)
  - δ(s,y)≤δ(s,u) (∵ y appears before u on a shortest path from s to u and all edge weights are non-negative)
  - y.d=δ(s,y)≤δ(s,u)≤u.d (by the upper-bound property)
  - But because both vertices u and y were in V-S when u was chosen in line 5, u.d≤y.d
  - y.d=δ(s,y)=δ(s,u)=u.d
  - Consequently u.d=δ(s,u), which contradicts our choice of u.
- u.d=δ(s,u) when u is added to S, and that this equality is maintained at all times thereafter

- Termination : At termination, Q={} which, along with our earlier invariant that Q=V-S, implies that S=V. u.d=δ(s,u) for all vertices u ∈V

Shortest-Path Algorithms

	Bellman-Ford	Dijkstra
Negative Edge	O	X
Positive Cycle	O	O
Negative Cycle	X	X
Time Complexity	O(|V||E|)	Array: $O(\|V\|^2)$ Min-heap: O((|V|+|E|)lg|V|) Fibonacci heap: O(|V|lg|V|+|E|)