Dijkstra's Algorithm

Single Source Shortest Path

Introduction

- Consider the problem of finding shortest paths between all pairs of vertices in a graph
- Problem might arise in making a table of distances between all pairs of cities for a road atlas
- given a weighted, directed graph $G = (V, E)$ with a weight function $w: E \to R$ that maps edges to real-valued weights

Introduction

- We wish to find, for every pair of vertices u, v ∈ V , a shortest (least–weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges
- We typically want the output in tabular form:
- the entry in u’s row and v’s column should be the weight of a shortest path from u to v

Introduction

- We can solve an all–pairs shortest–paths problem by running a single–source shortest–paths algorithm |V| times, once for each vertex as the source.
- If all edge weights are nonnegative, we can use Dijkstra's algorithm

Introduction

- If the graph has negative-weight edges, we cannot use Dijkstra's algorithm
- Instead, we must run the slower Bellman–Ford algorithm once from each vertex
- The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$

Dijkstra's algorithm

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

Dijkstra's algorithm

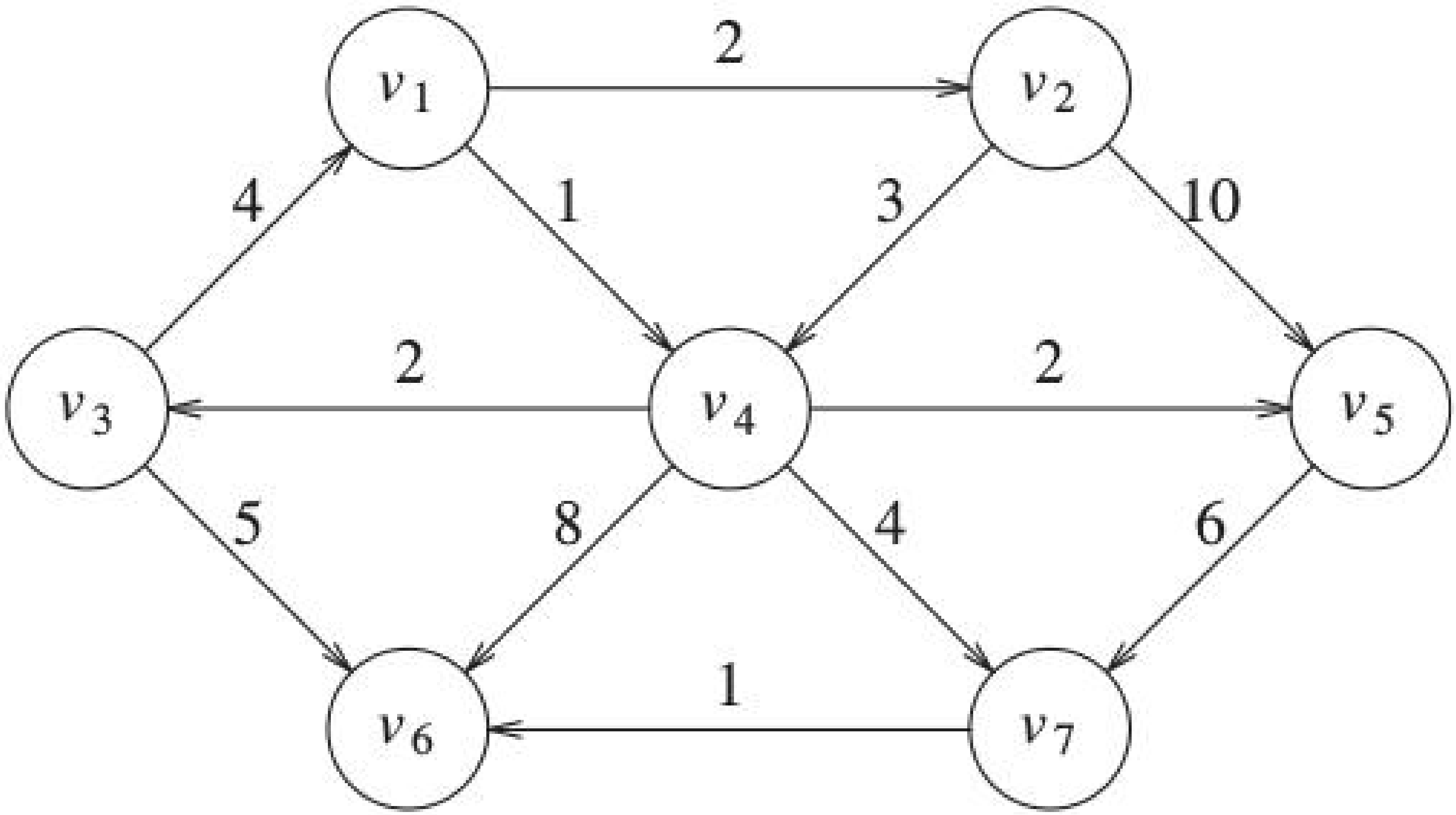


Figure 9.20 The directed graph G (again)

Dijkstra's algorithm

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra's algorithm

Dijkstra’s algorithm

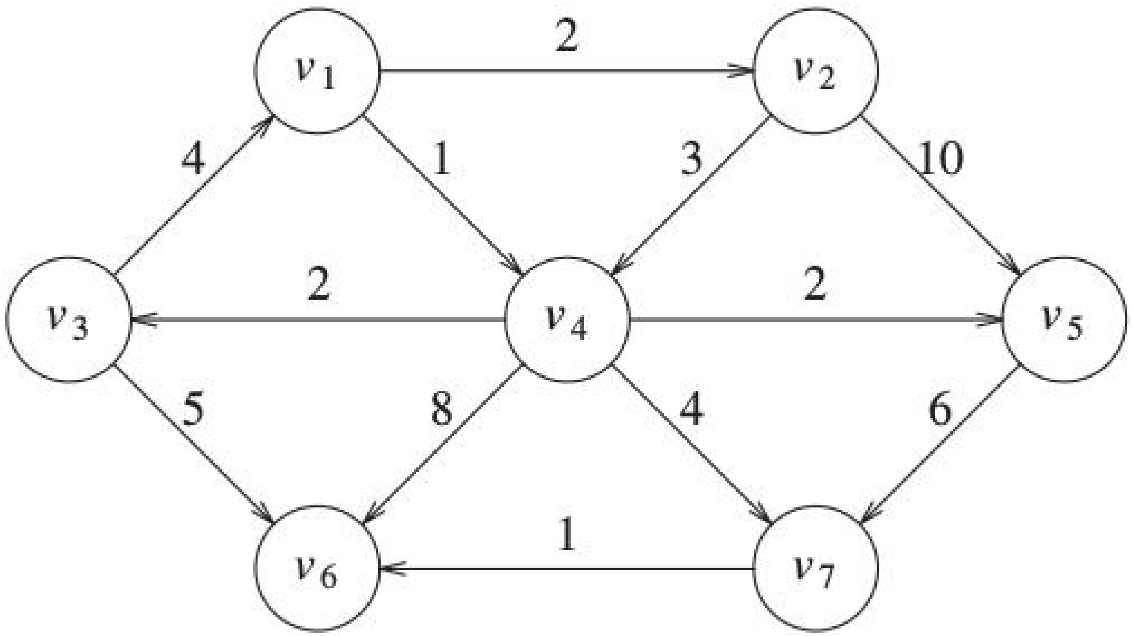


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22 After v_1 is declared *known*

Dijkstra's algorithm

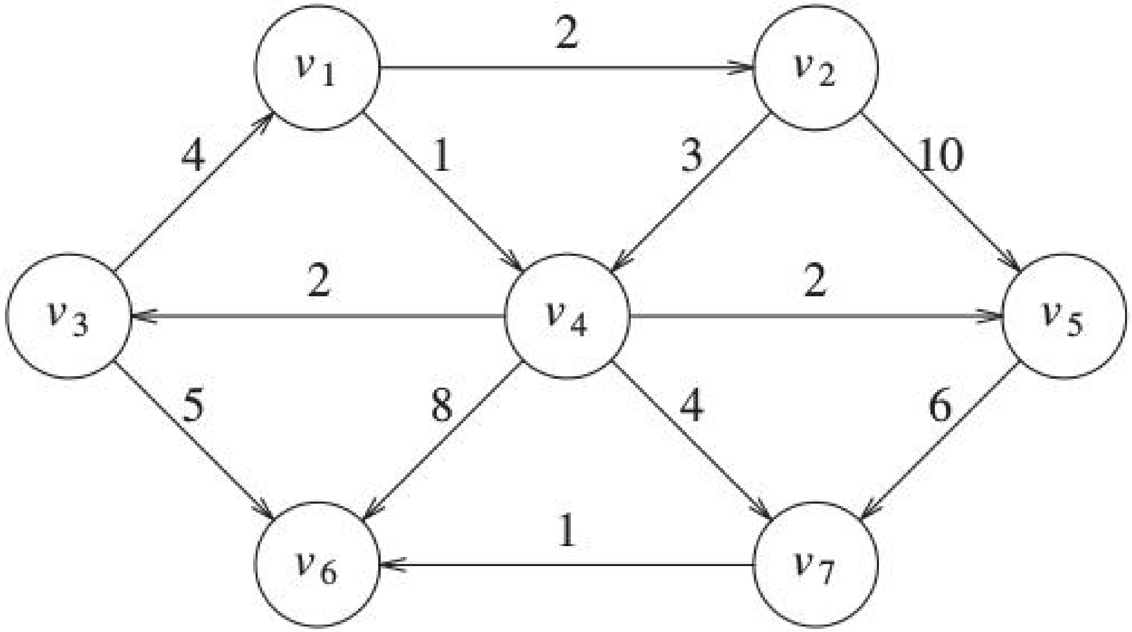


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.23 After v_4 is declared *known*

Dijkstra's algorithm

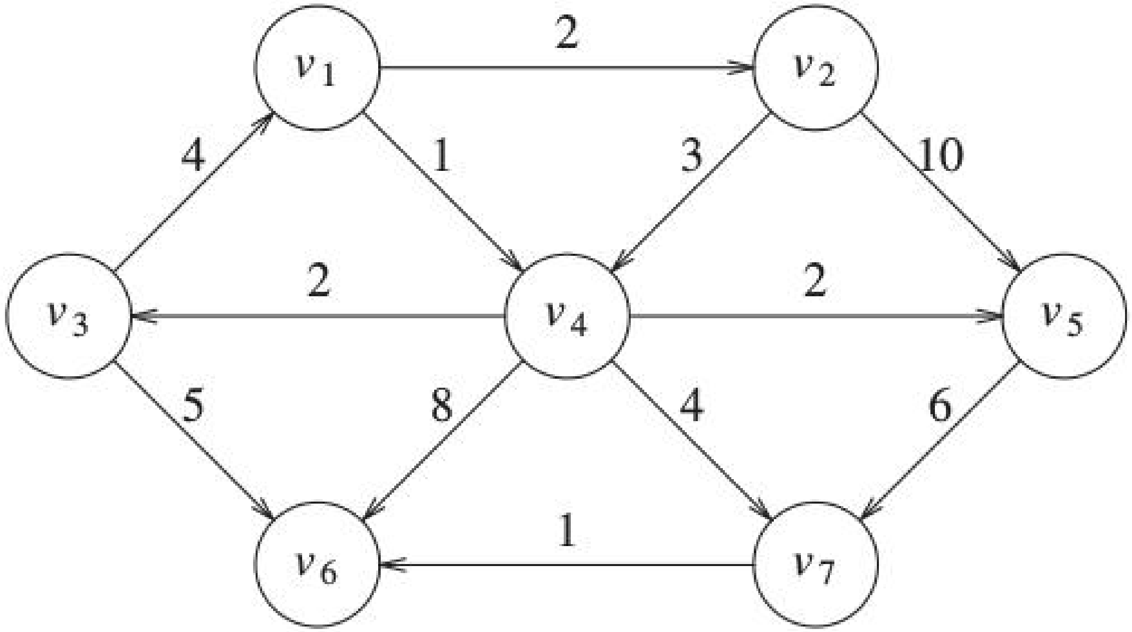


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.24 After v_2 is declared *known*

Dijkstra's algorithm

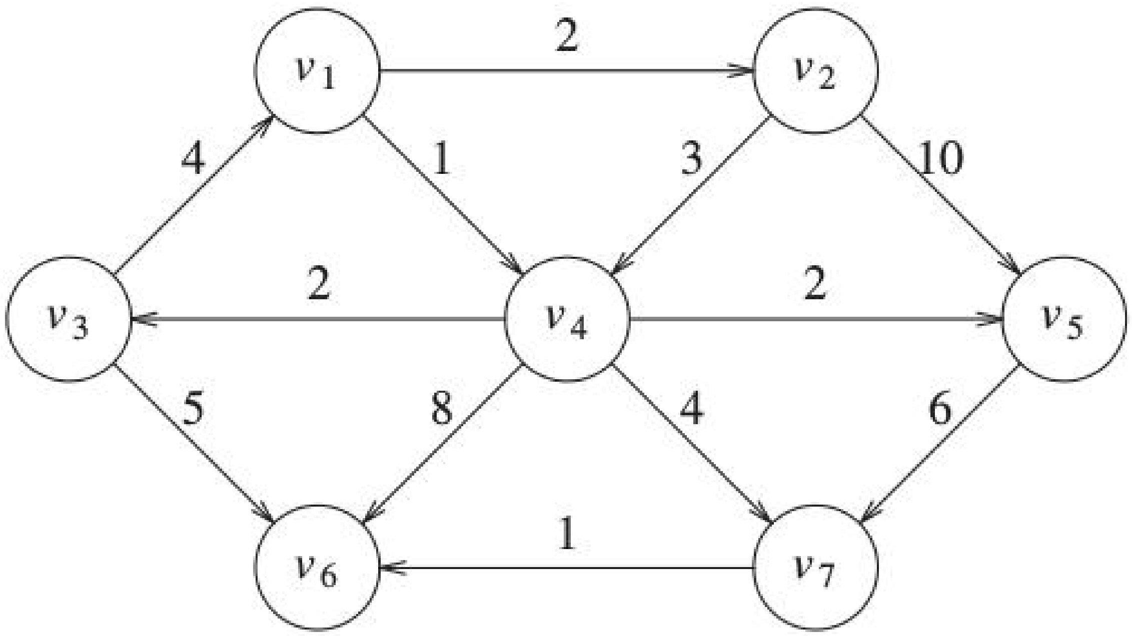


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

Figure 9.25 After v_5 and then v_3 are declared *known*

Dijkstra's algorithm

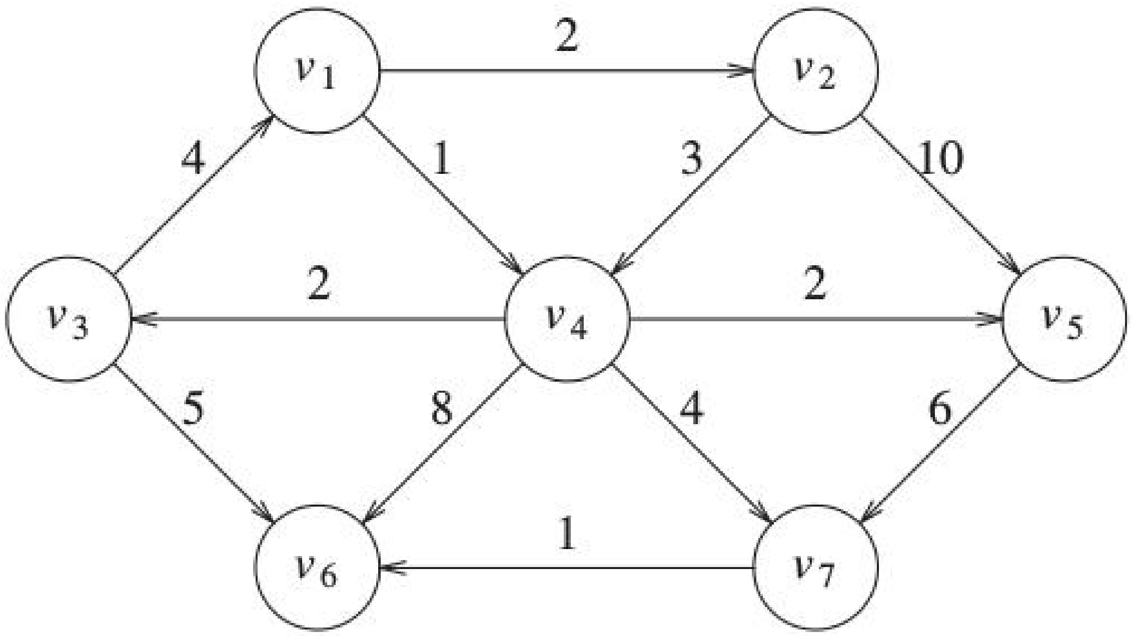


Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Figure 9.26 After v_7 is declared *known*

Dijkstra's algorithm

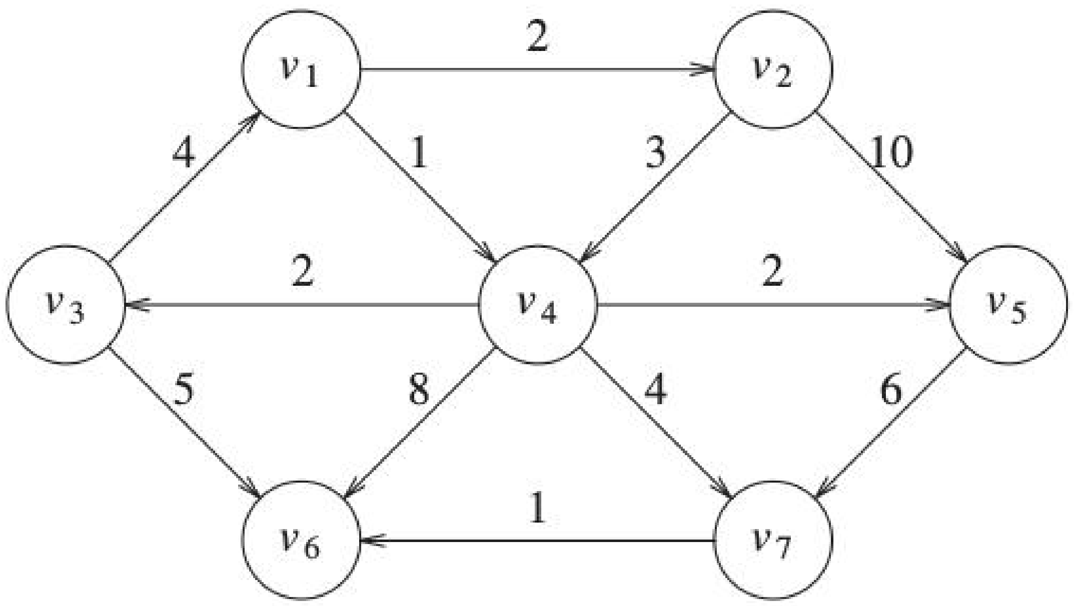
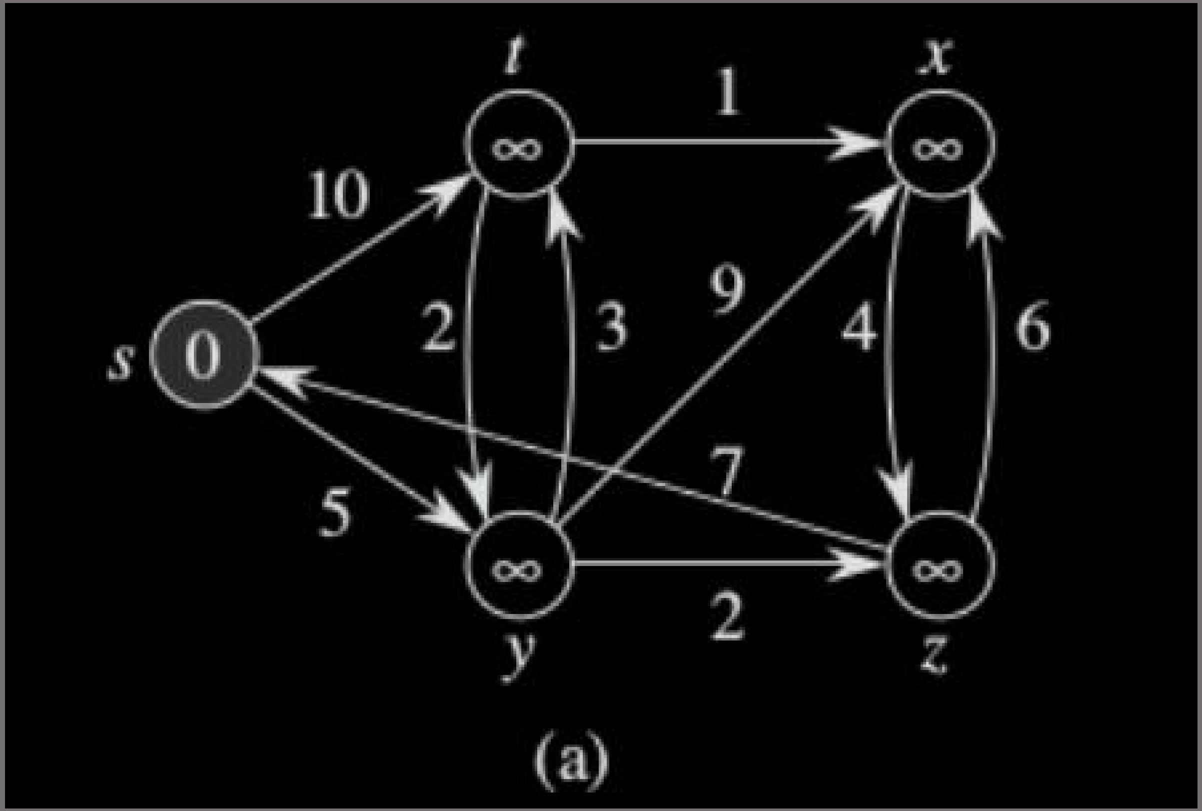


Figure 9.20 The directed graph G (again)

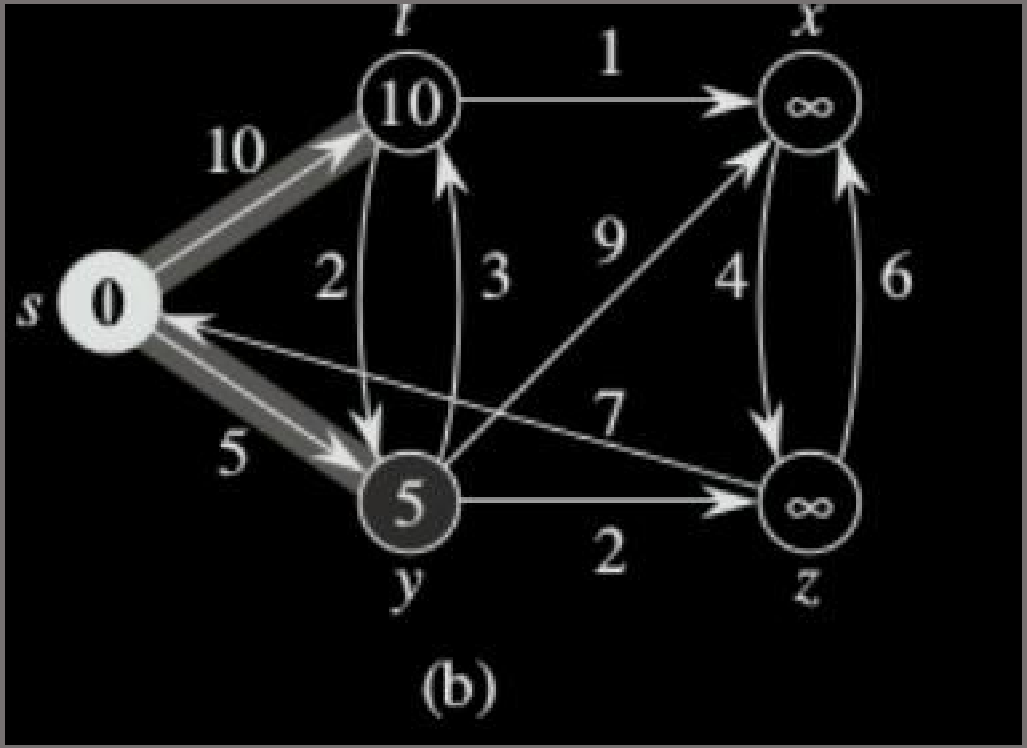
v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Figure 9.27 After v_6 is declared *known* and algorithm terminates

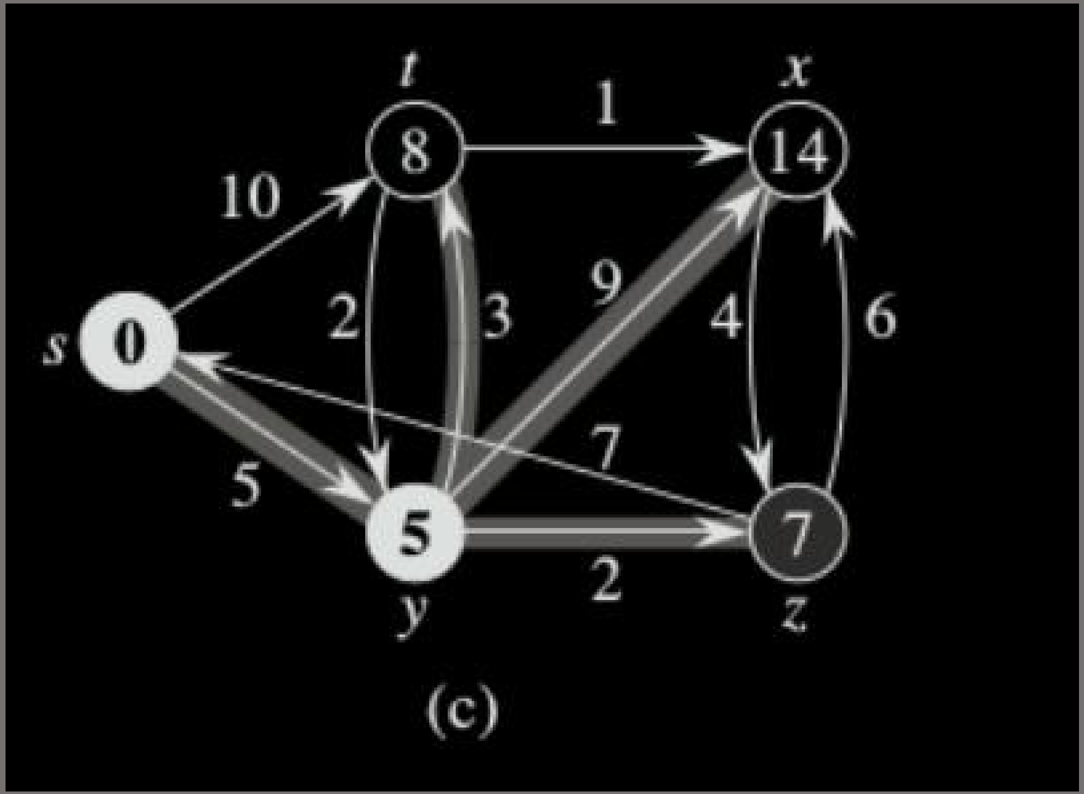
Dijkstra's algorithm



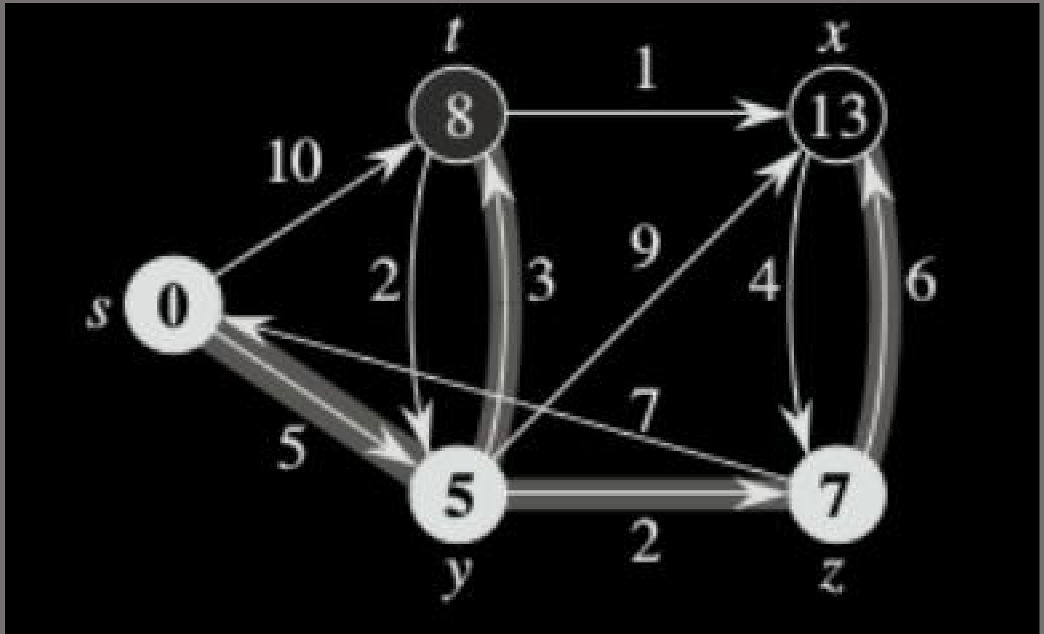
Dijkstra's algorithm



Dijkstra's algorithm

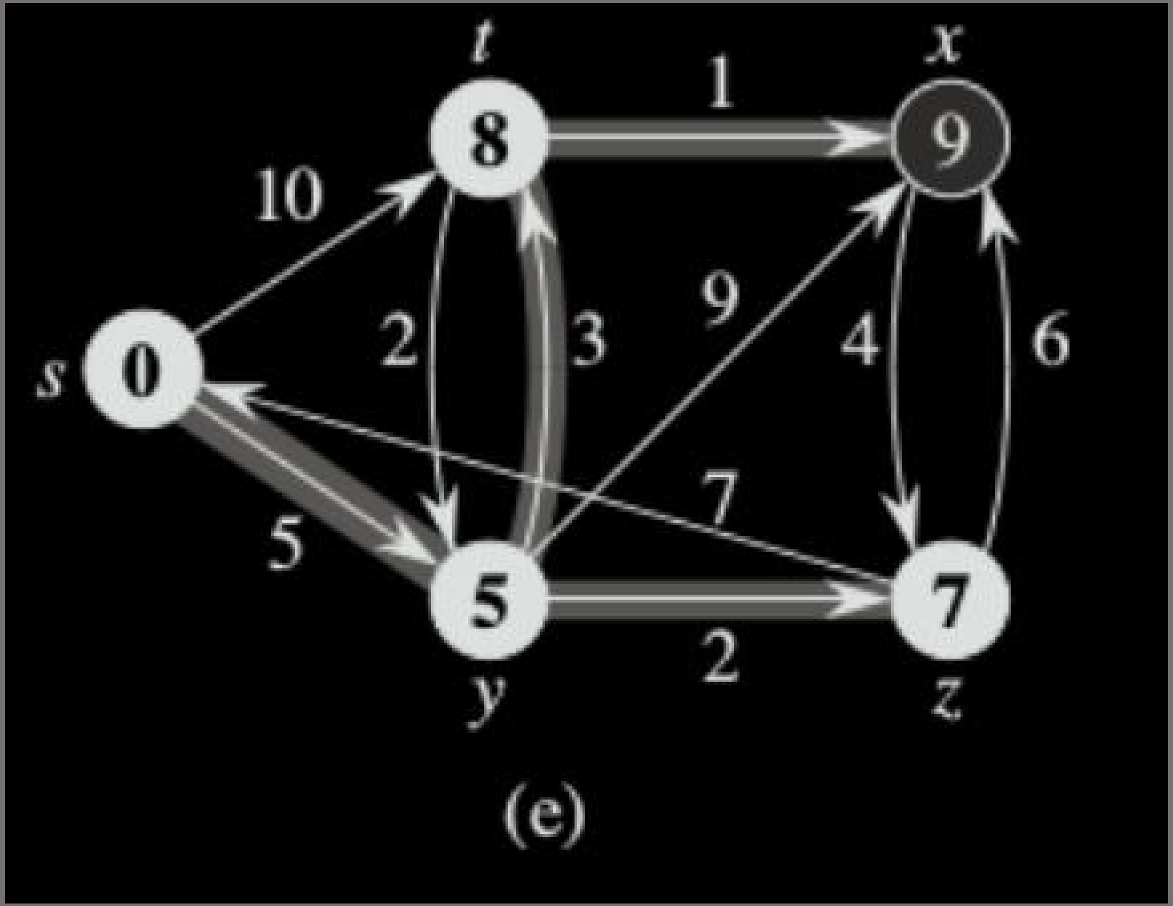


Dijkstra's algorithm

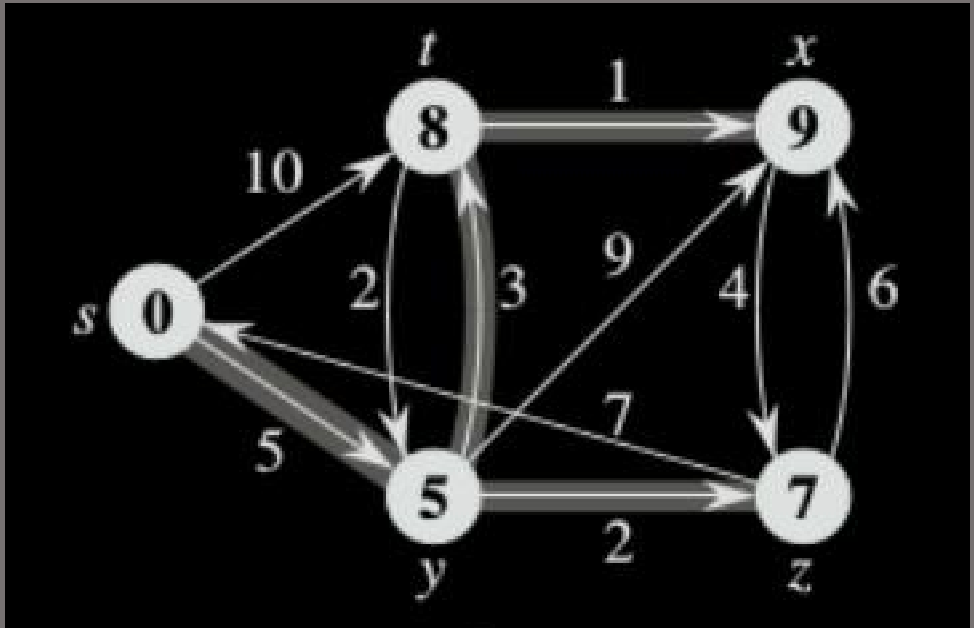


(d)

Dijkstra's algorithm



Dijkstra’s algorithm



(f)

Analysis

- Because each vertex u ∈ V is added to set S exactly once, each edge in the adjacency list Adj[u] is examined in the for loop of lines 7 – 8 exactly once during the course of the algorithm.
- Since the total number of edges in all the adjacency lists is |E|, this for loop iterates a total of |E| times, and thus the algorithm calls DECREASE–K EY at most |E| times overall

Analysis

- running time of Dijkstra's algorithm depends on how we implement the min–priority queue
- Case I – simply store v.d in the v^{th} entry of an array
- Each INSERT and DECREASE–K EY operation takes O(1) time, and each E XTRACT–M IN operation takes O(V) time
- total time of $O(V^2 + E) = O(V^2)$

Analysis

- Case II – we can improve the algorithm by implementing the min–priority queue with a binary min–heap
- Each EXTRACT–MIN operation then takes time O(lg V)
- time to build the binary min–heap is O(V)
- Each DECREASE –KEY operation takes time O(lg V), and there are still at most |E| such operations
- total running time is therefore O(V + E) lg V) which is O.E lg V / if all vertices are reachable from the source

Analysis

- Case III – achieve a running time of O(V lg V + E) by implementing the min-priority queue with a Fibonacci heap
- The amortized cost of each of the |V| EXTRACT –M IN operations is O(lg V), and each DECREASE –K EY call, of which there are at most |E|, takes only O(1) amortized time

Dijkstra’s algorithm

- We allow negative–weight edges, but we assume for the time being that the input graph contains no negative–weight cycles.
- Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min–priority queue it drops down to $O(V + E \log V)$

Bellman–Ford algorithm

- Solves the single–source shortest–paths problem in the general case in which edge weights may be negative
- Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman–Ford algorithm returns a boolean value indicating whether or not there is a negative–weight cycle that is reachable from the source.
- If there is such a cycle, the algorithm indicates that no solution exists.

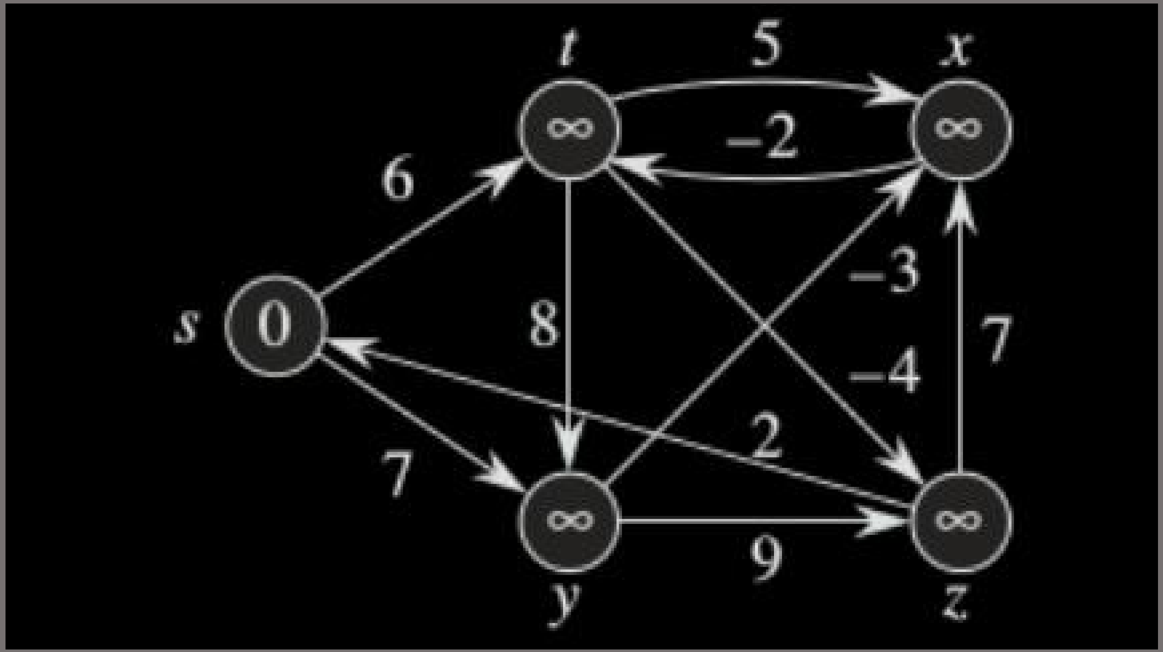
Bellman–Ford algorithm

- No such cycle, the algorithm produces shortest paths and their weights
- 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 $\delta(s,v)$
- The algorithm returns TRUE if and only if the graph contains no negative–weight cycles that are reachable from the source.

Bellman–Ford algorithm

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

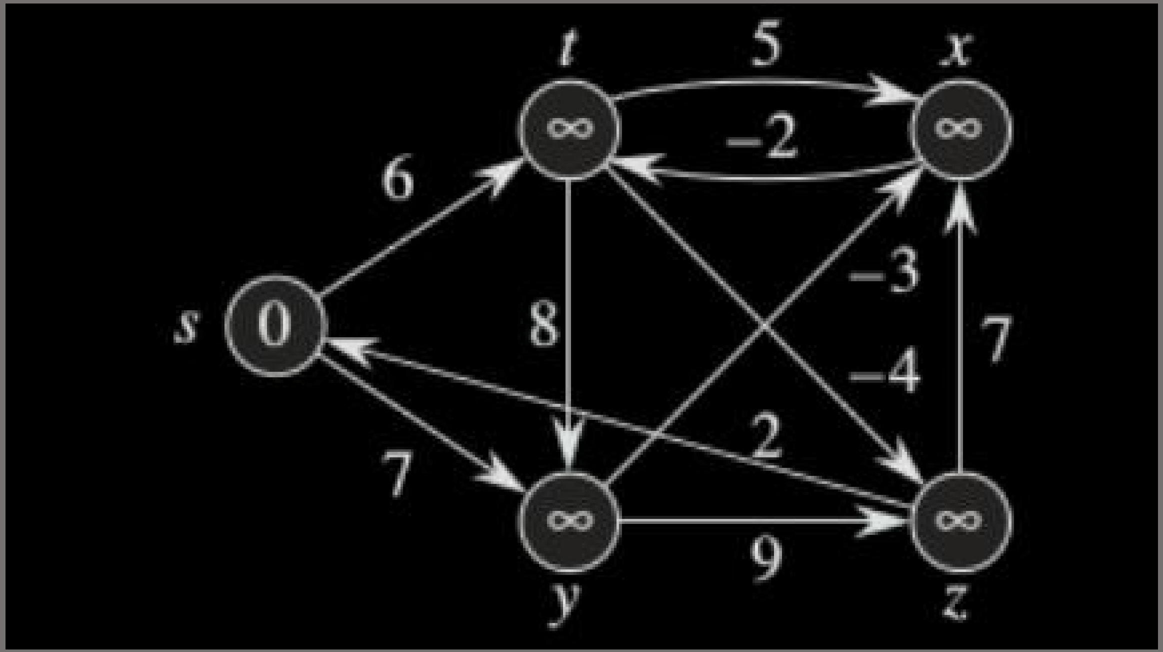
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	
x	∞	∞
y	∞	
z	∞	

(t, x)									

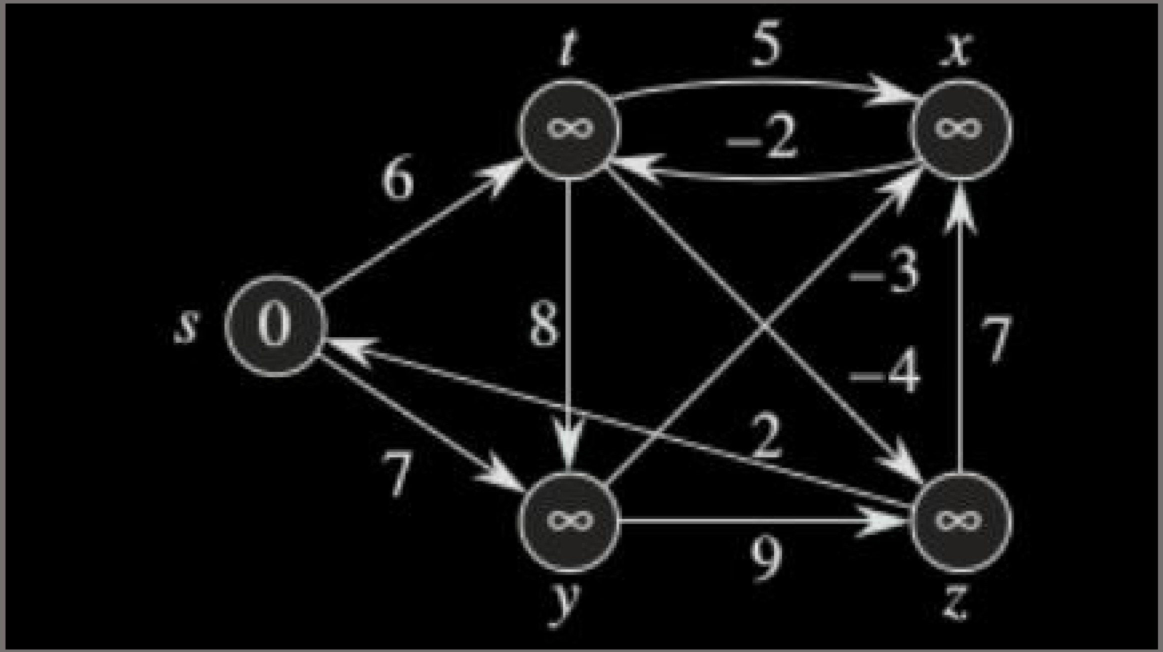
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	
x	∞	∞
y	∞	∞
z	∞	

(t, x)	(t, y)								

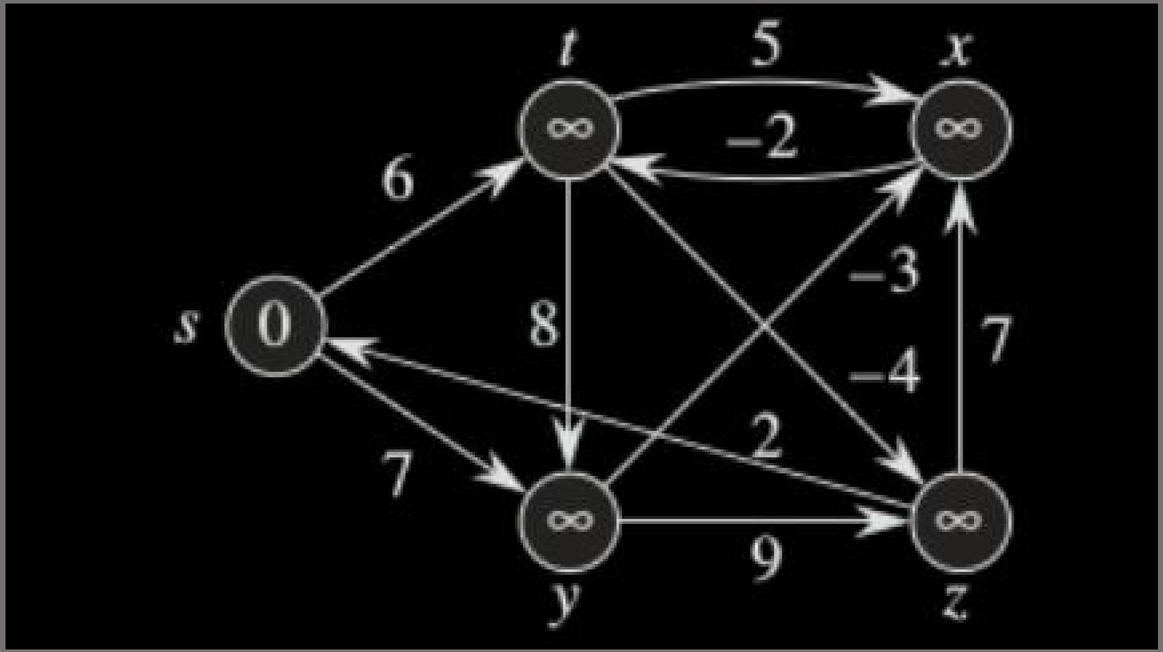
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)							

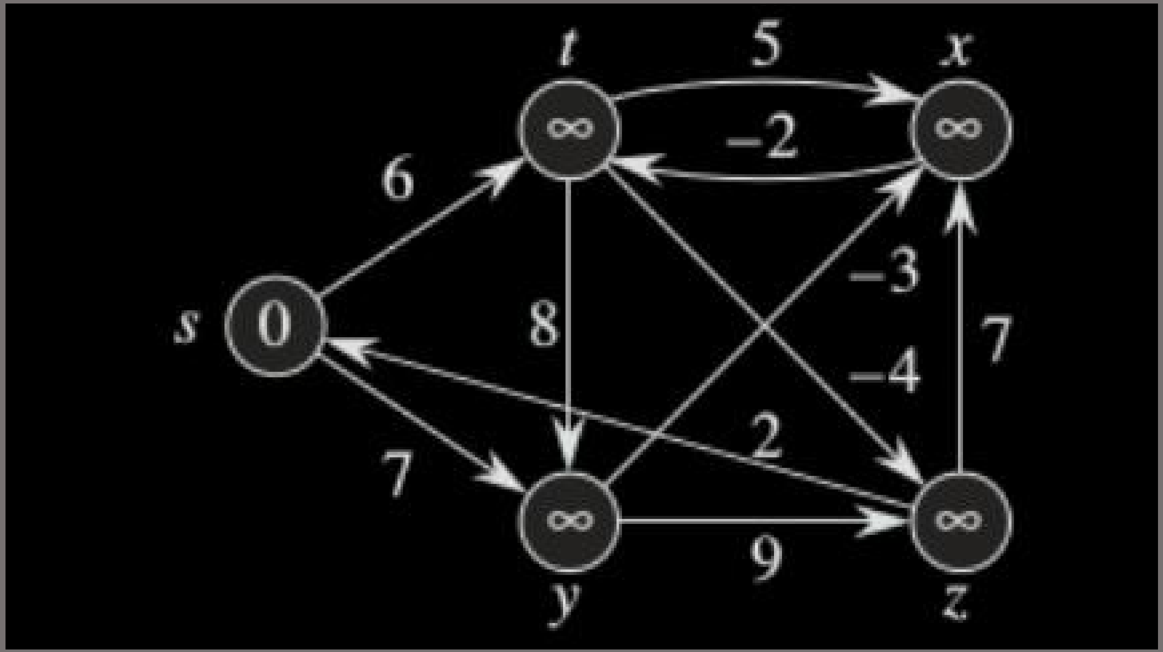
Bellman–Ford algorithm



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	∞
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)						

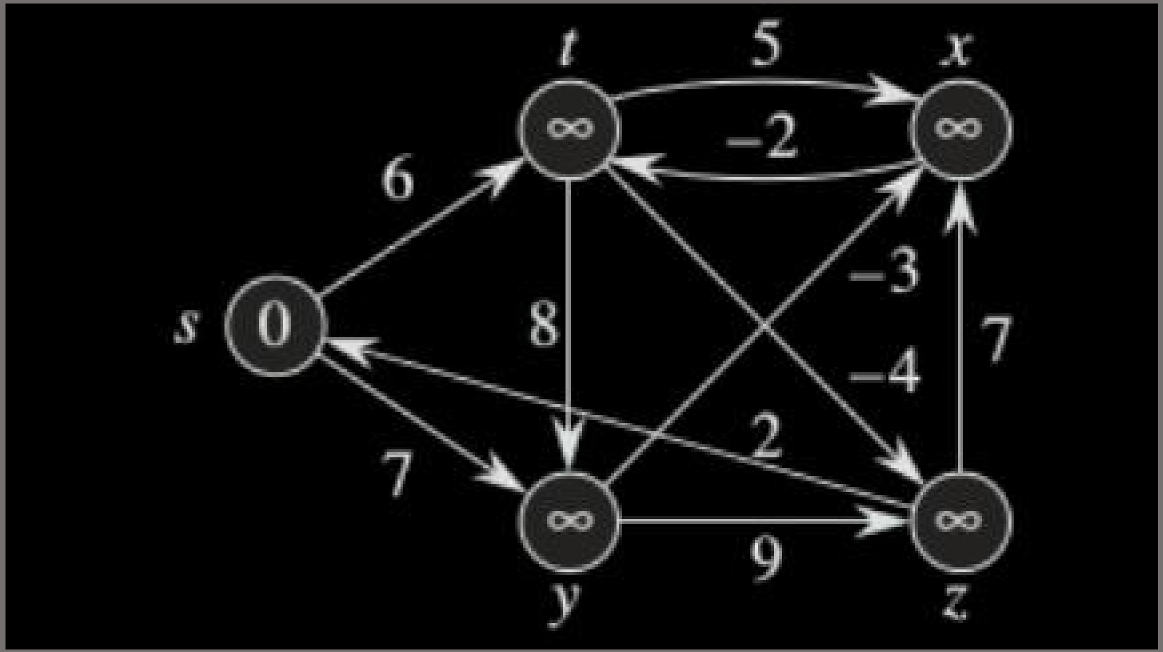
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	∞
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)					

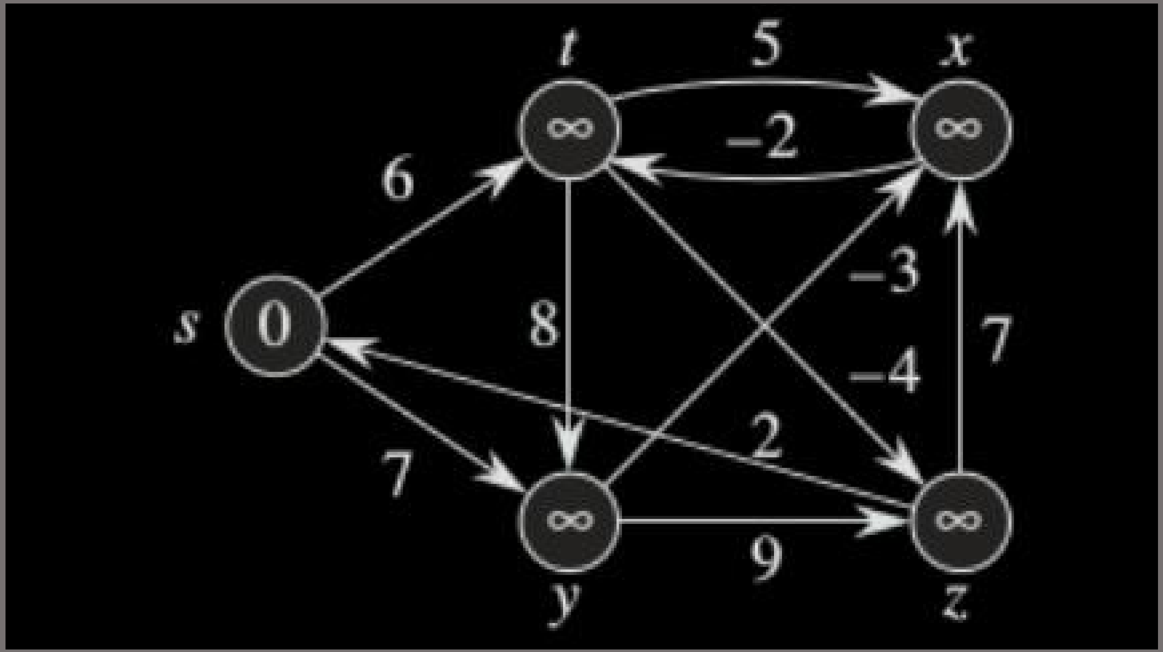
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	∞
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)				

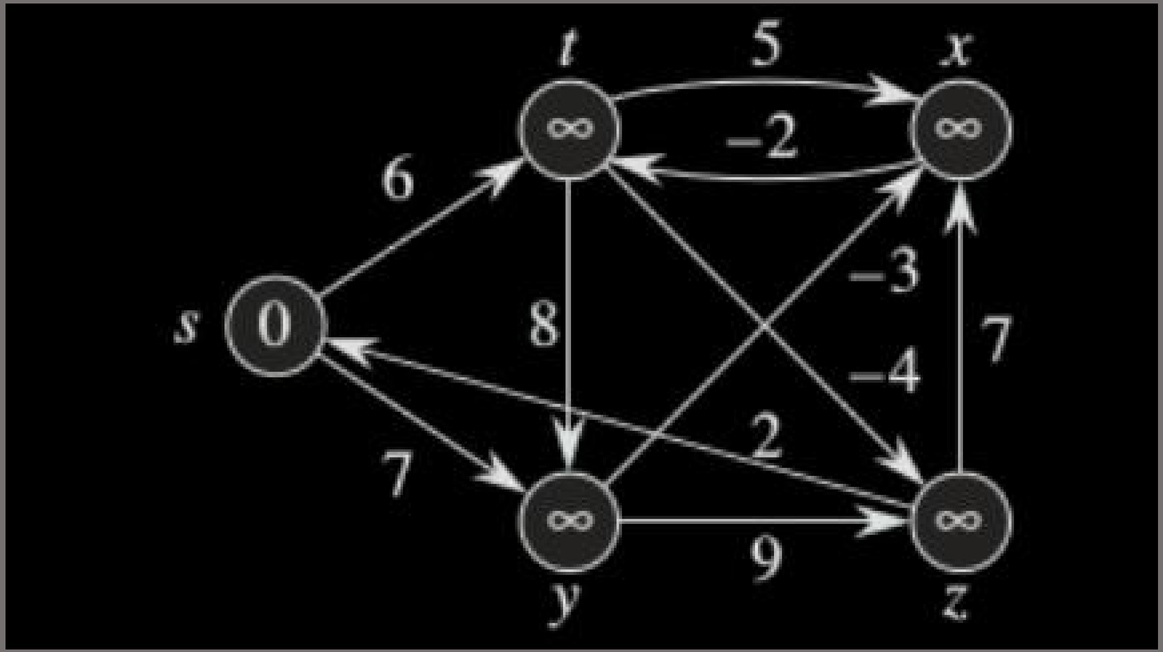
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	∞
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)			

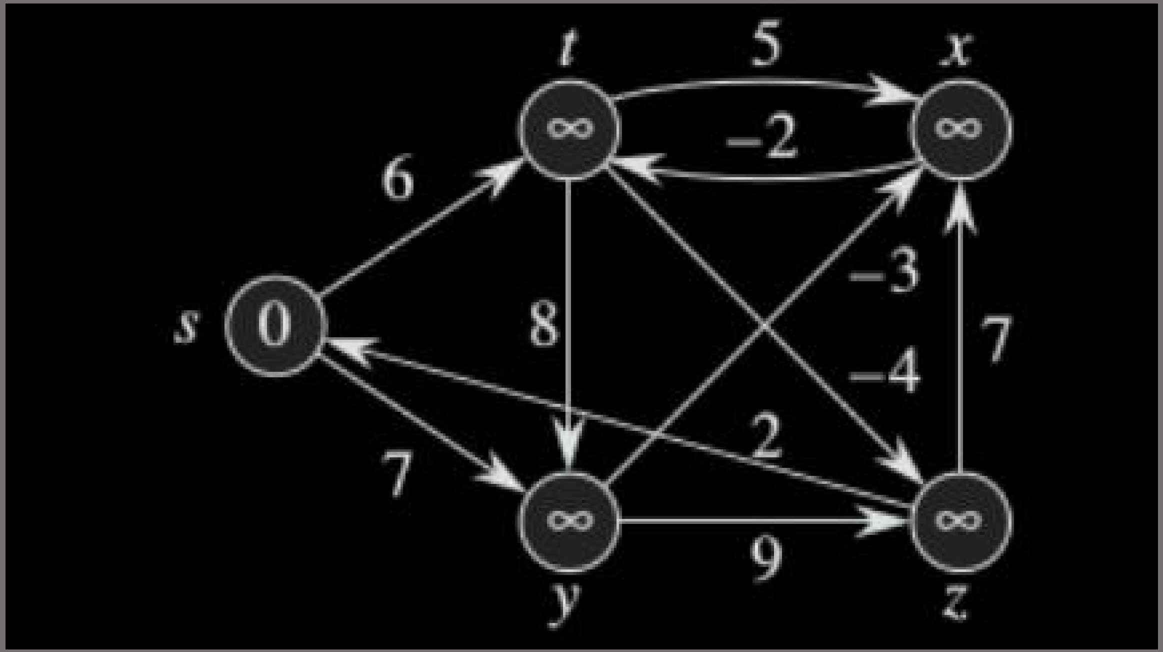
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	∞
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)		

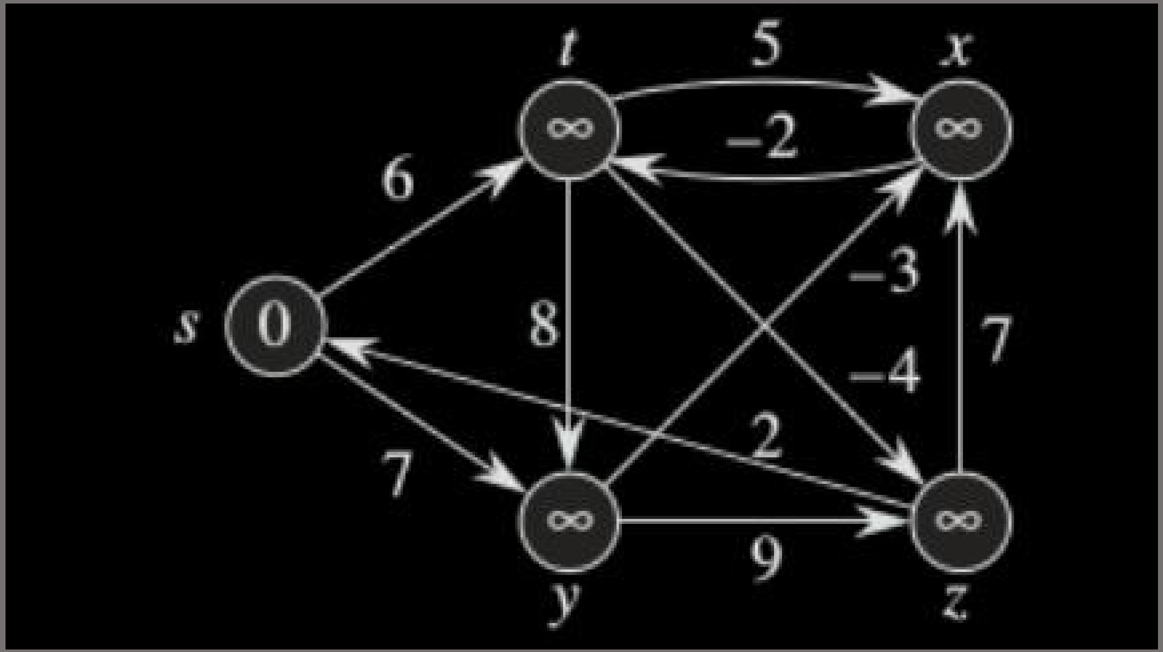
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	6
x	∞	∞
y	∞	∞
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	

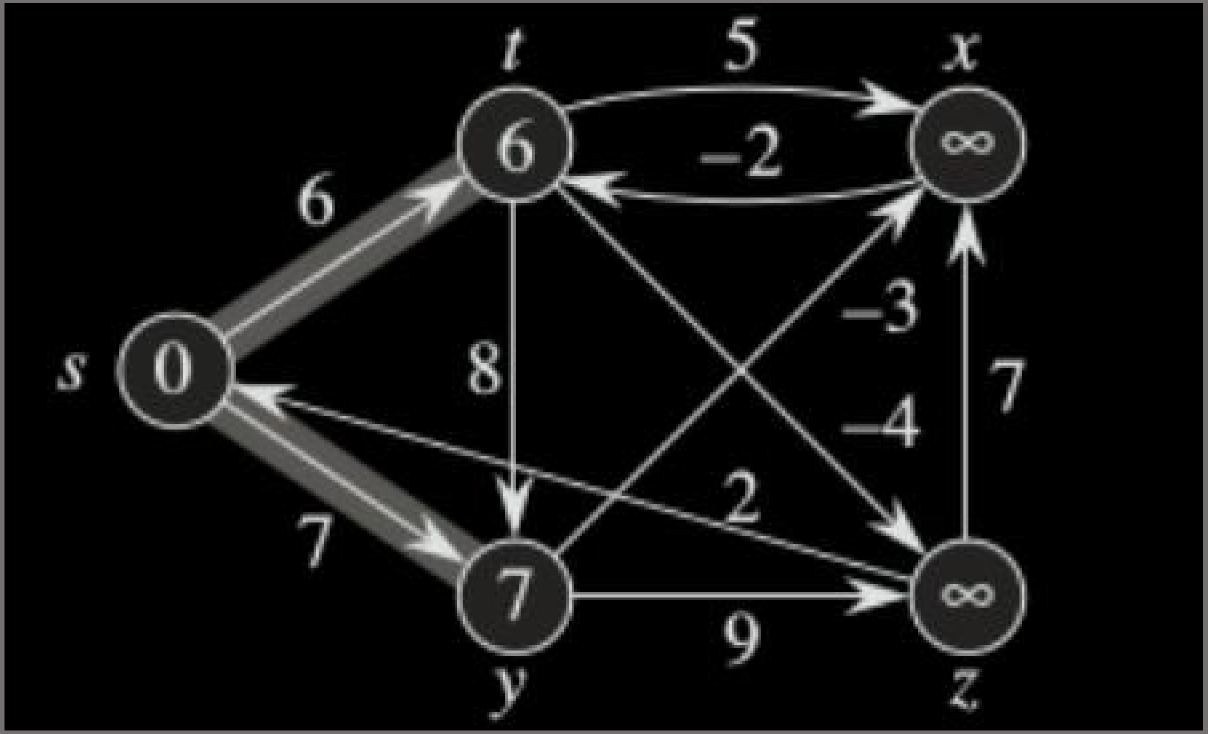
Bellman–Ford algorithm – Iteration 1



Edge Name	Old Cost	Updated Cost
s	0	0
t	∞	6
x	∞	∞
y	∞	7
z	∞	∞

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

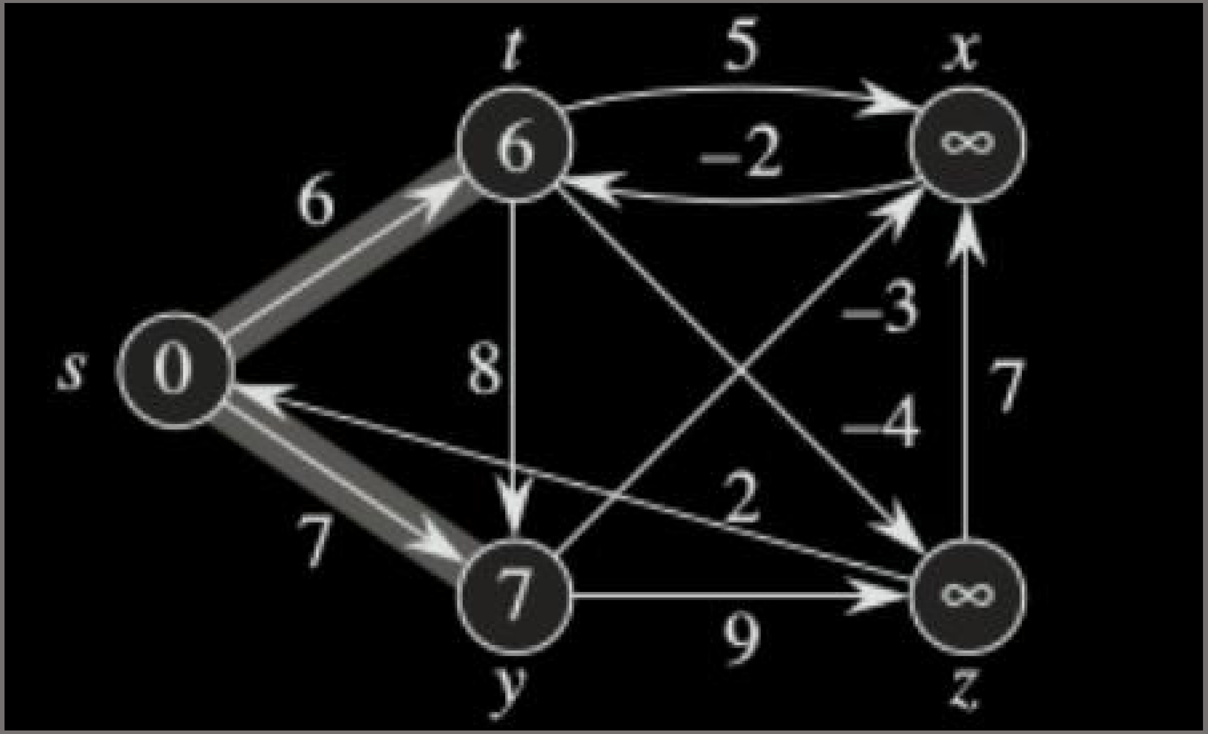
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

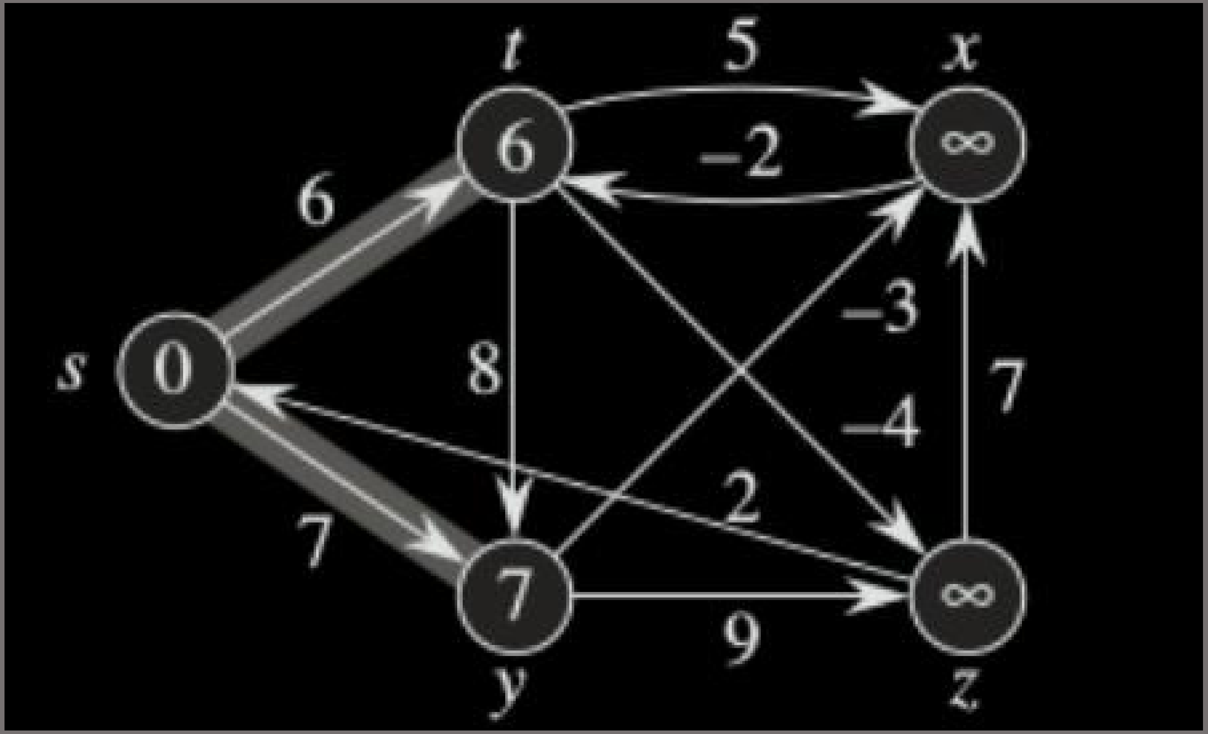
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

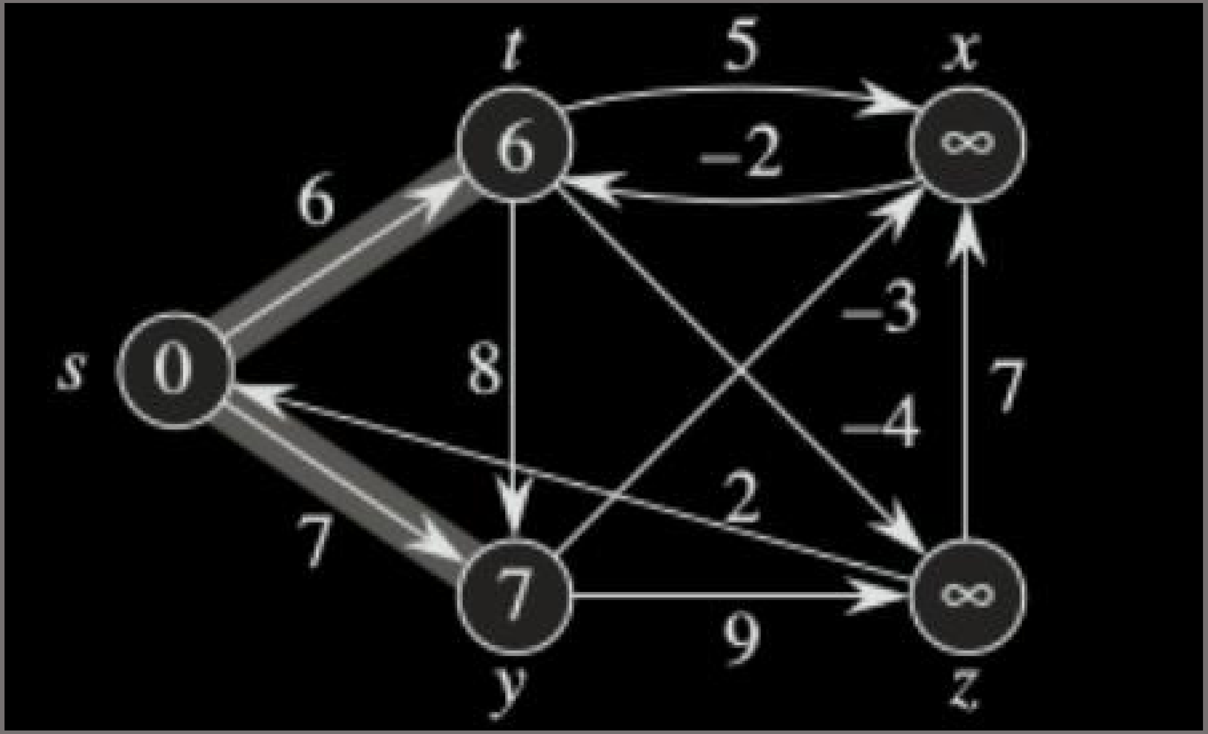
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

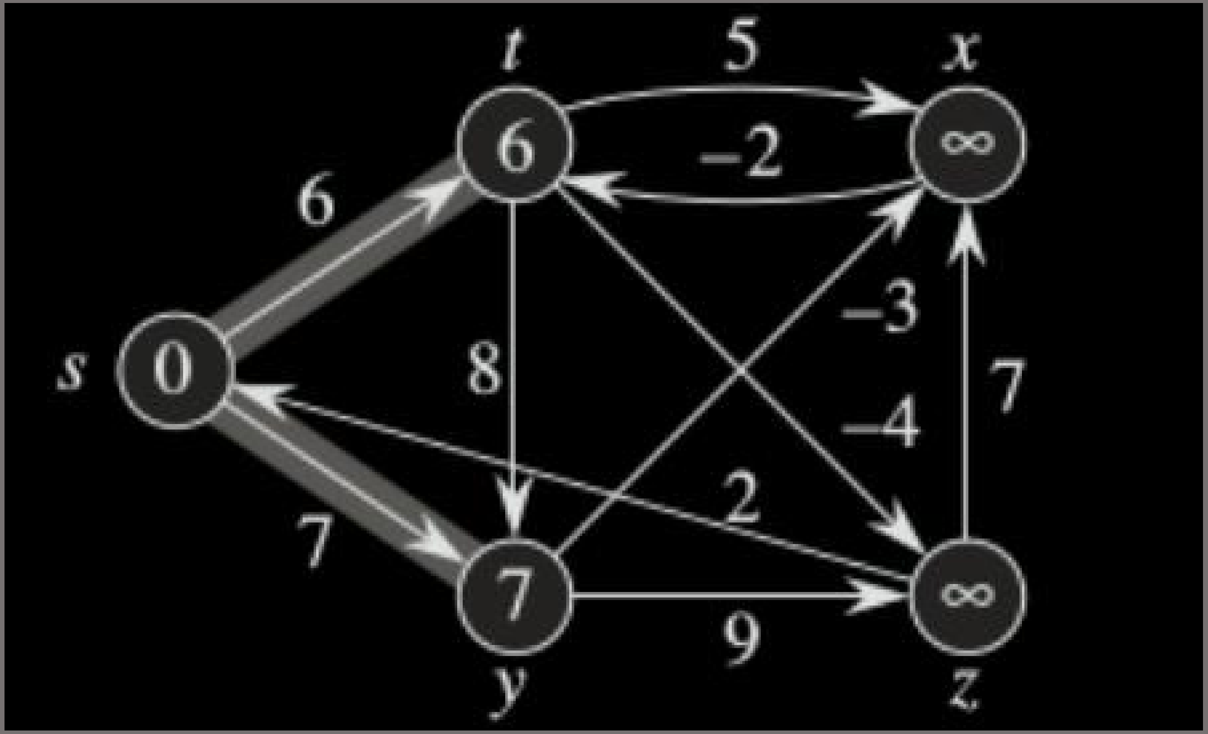
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

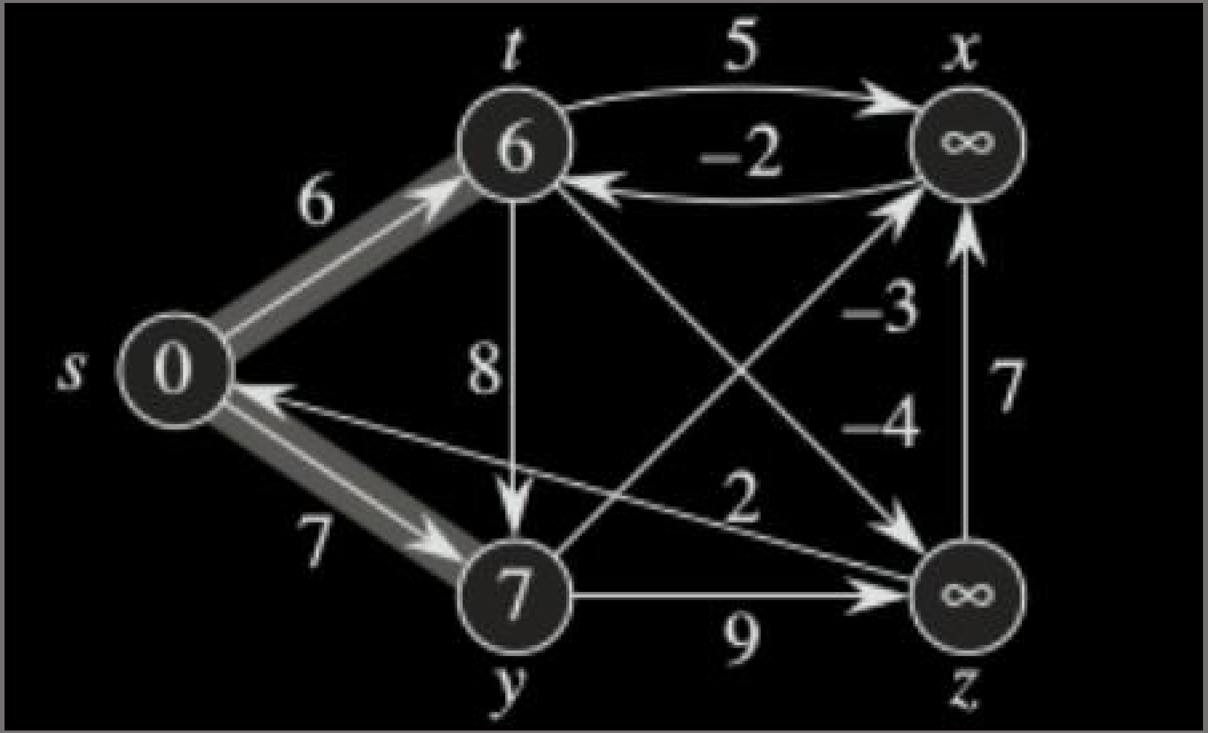
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

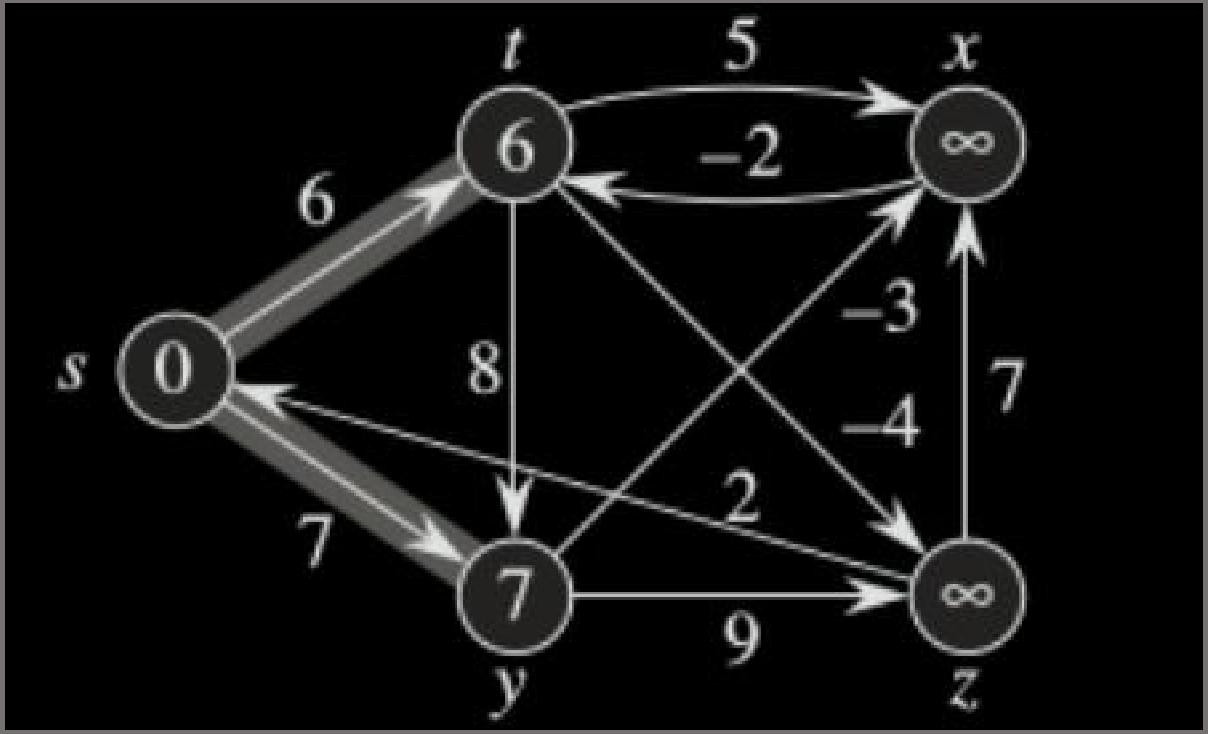
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

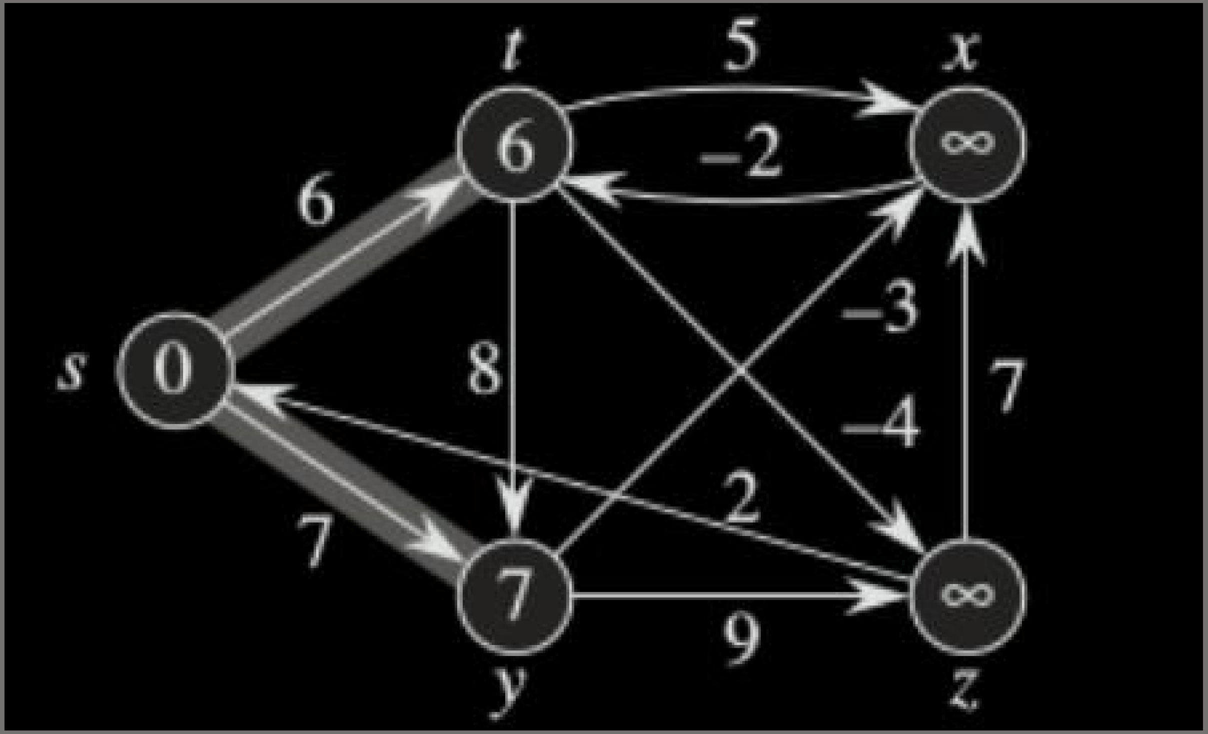
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

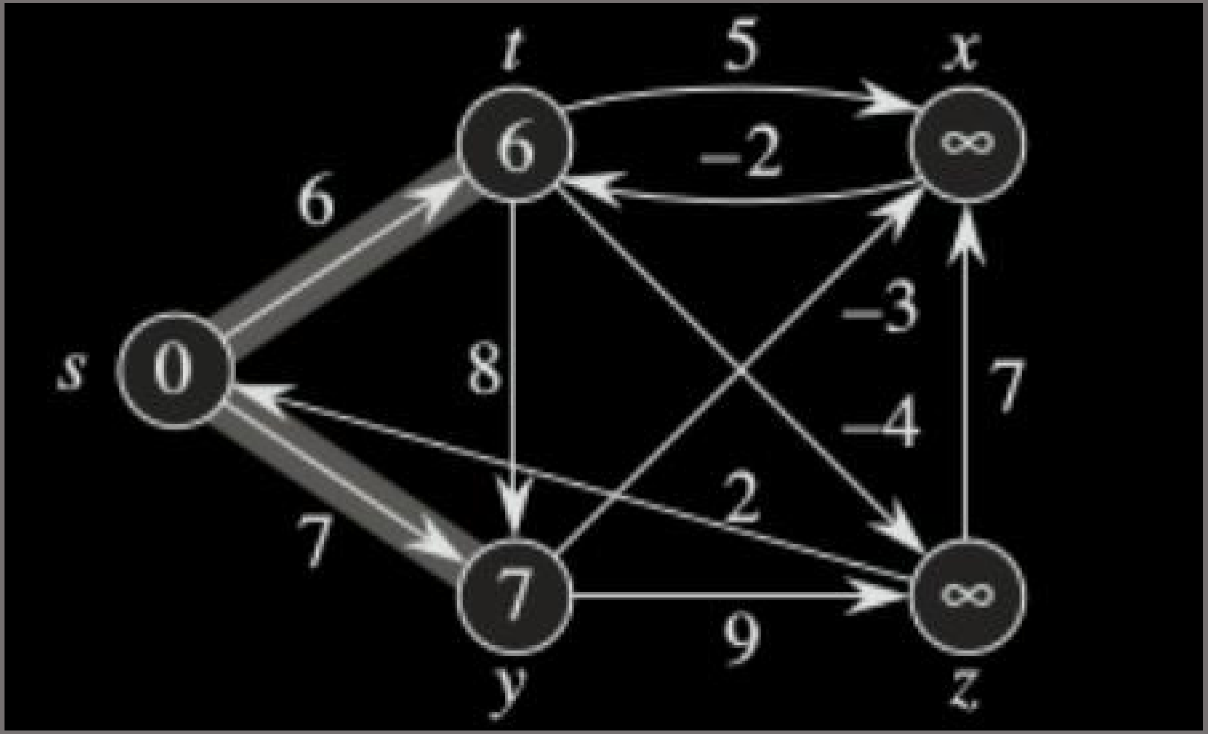
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

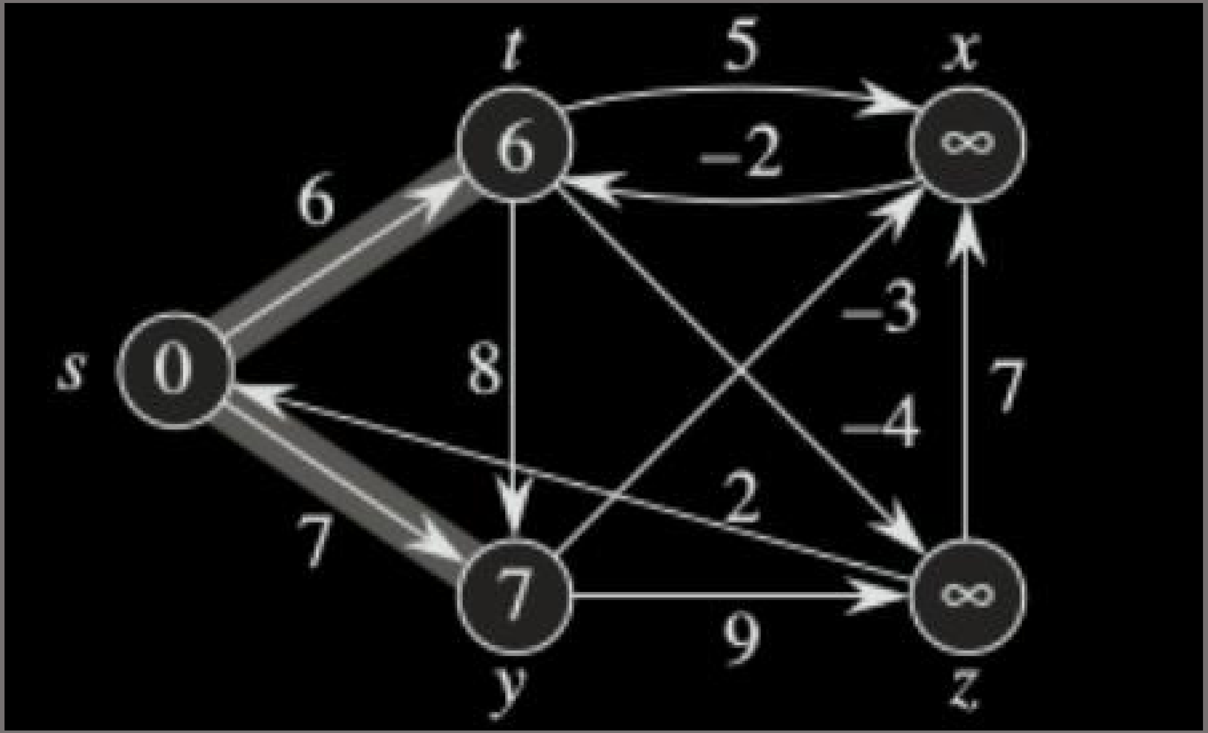
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

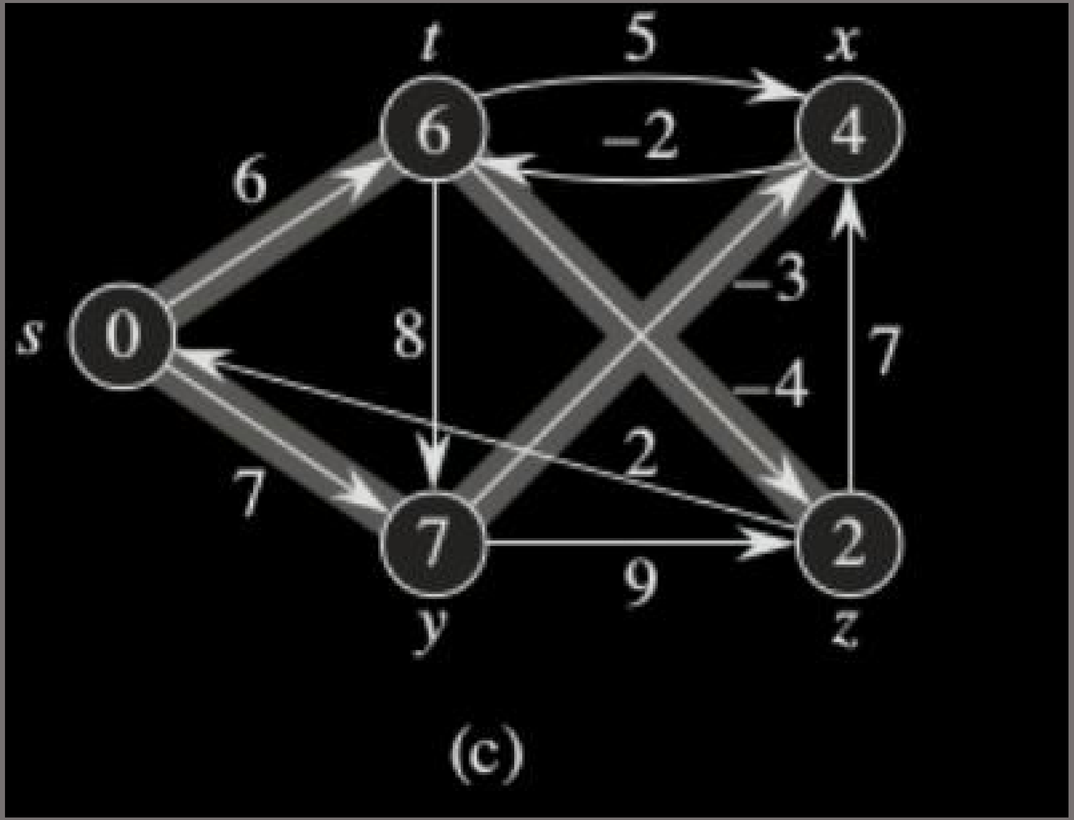
Bellman–Ford algorithm – Iteration 2



Edge Name	Old Cost	Updated Cost
s	0	
t	6	
x	∞	
y	7	
z	∞	

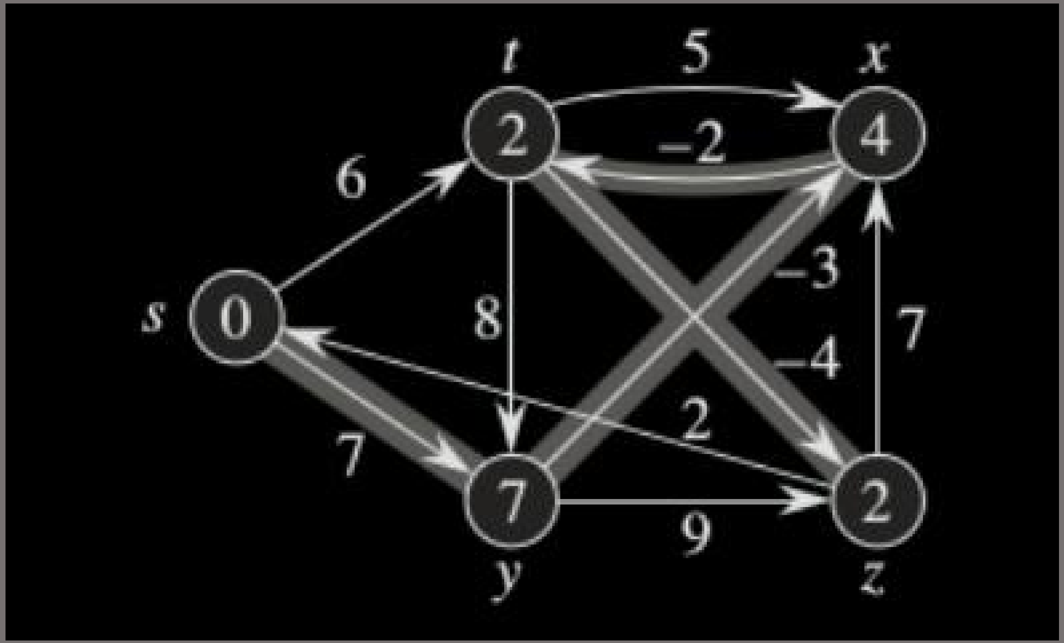
(t, x)	(t, y)	(t, z)	(x, t)	(y, x)	(y,z)	(z, x)	(z,s)	(s,t)	(s,y)

Bellman–Ford algorithm



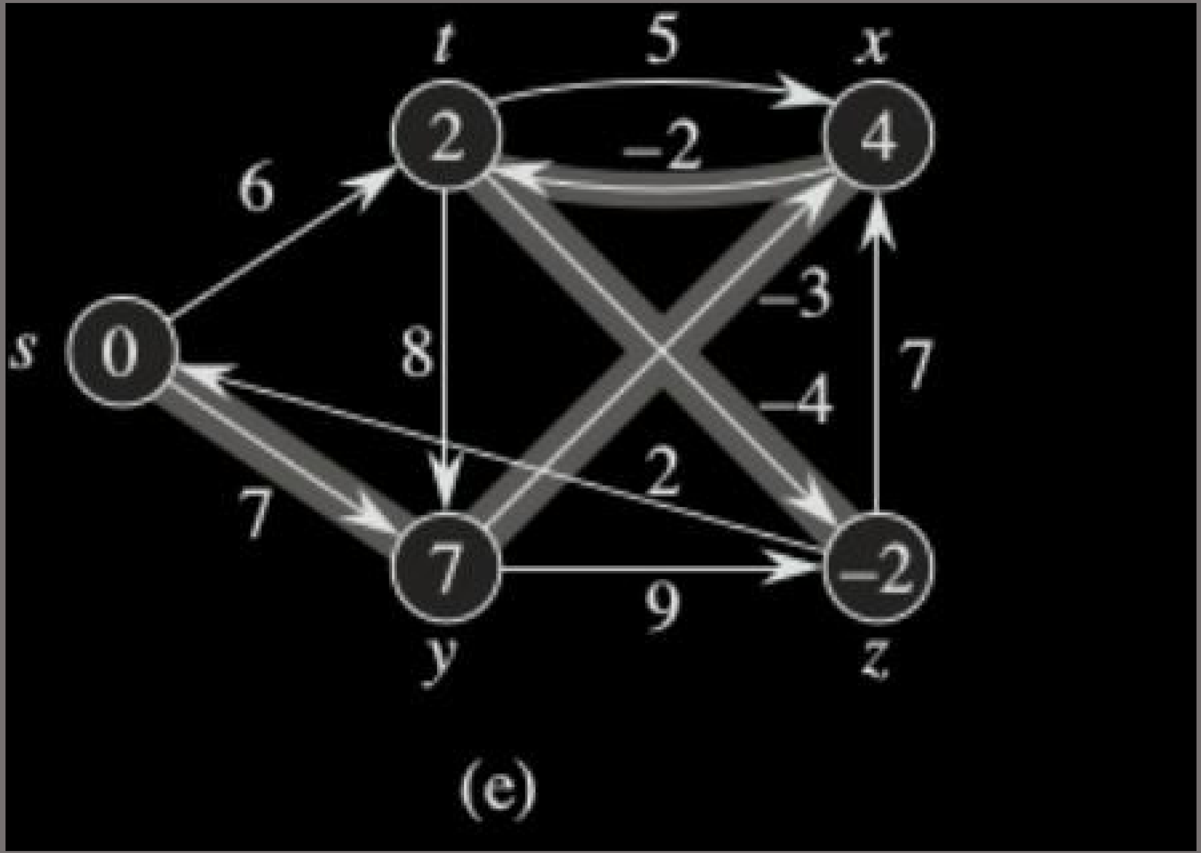
each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y,z), (z, x), (z,s), (s,t), (s,y)

Bellman–Ford algorithm



each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y,z), (z, x), (z,s), (s,t), (s,y)

Bellman–Ford algorithm



each pass relaxes the edges in the order (t, x), (t, y), (t, z), (x, t), (y, x), (y,z), (z, x), (z,s), (s,t), (s,y)

Bellman–Ford algorithm

- time complexity of Bellman–Ford is O(VE), which is more than Dijkstra.