Graph Algorithms

Part 3 Shortest Path

CS 325

Shortest-Path Problems

- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
  - Unweighted shortest-paths – BFS.

Shortest Paths: Applications

- Flying times between cities
- Distance between street corners
- Cost of doing an activity
  - Vertices are states
  - Edge weights are costs of moving between states

Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

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

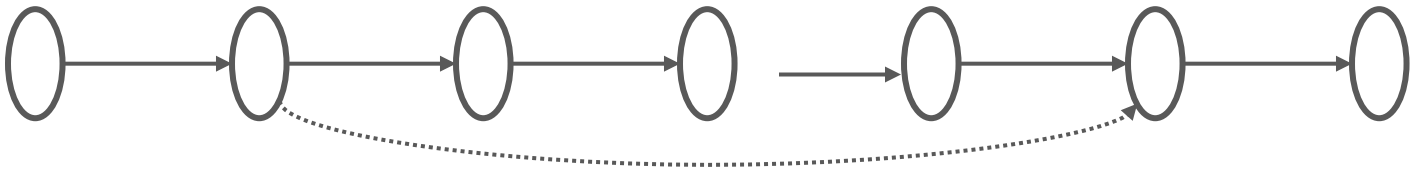
- Shortest path = a path of the minimum weight

Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" = minimum weight
  - Weight of path is sum of edges

Shortest Path Properties

- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:

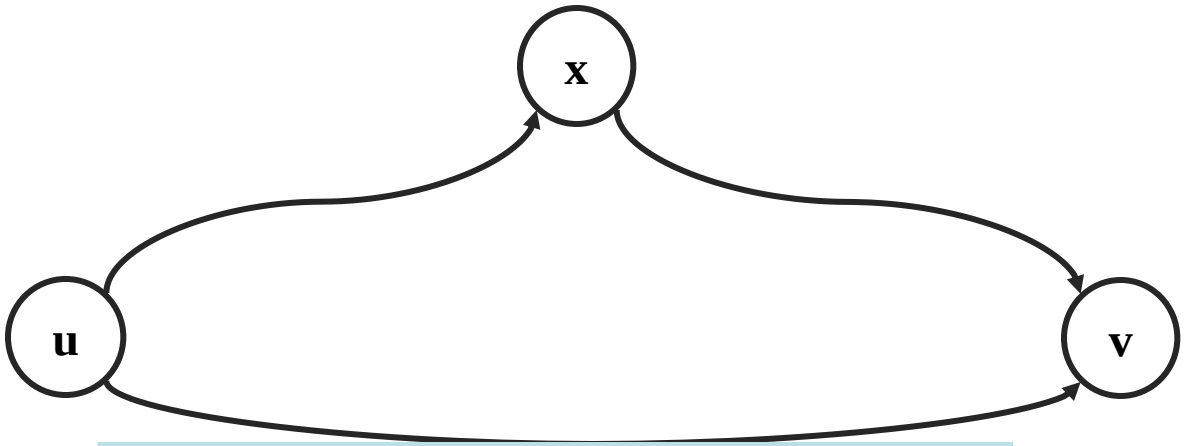


Proof: suppose some subpath is not a shortest path

- There must then exist a shorter subpath
- Could substitute the shorter subpath for a shorter path
- But then overall path is not shortest path. Contradiction

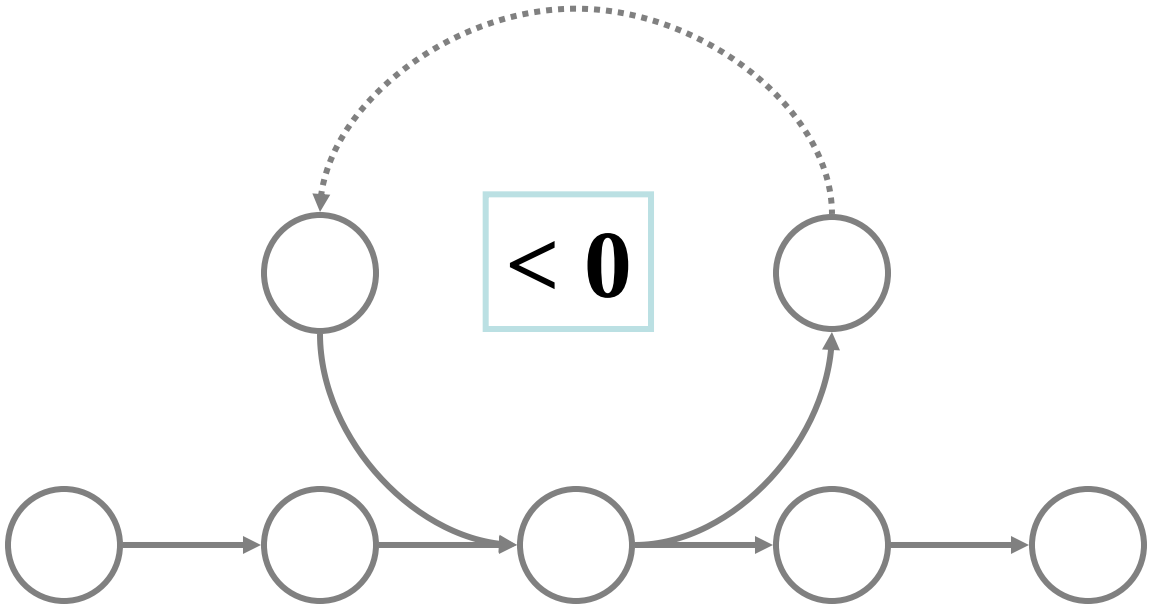
Shortest Path Properties

- Define $\delta(u,v)$ to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$



Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist



Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small "lengths" would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)

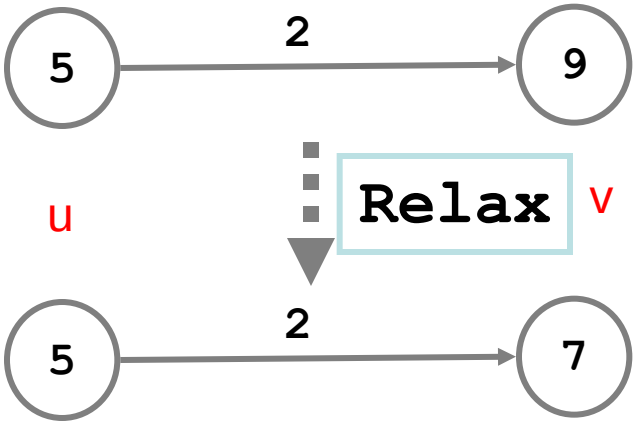
  Any shortest-path in graph *G* can be no longer than *n*–1 edges, where *n* is the number of vertices

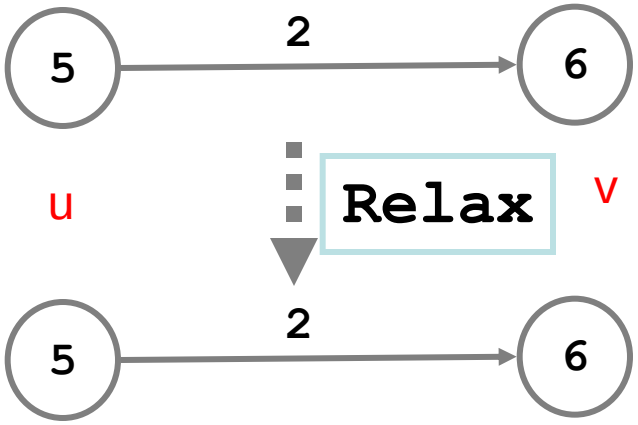
Relaxation

A key technique in shortest path algorithms is *relaxation*

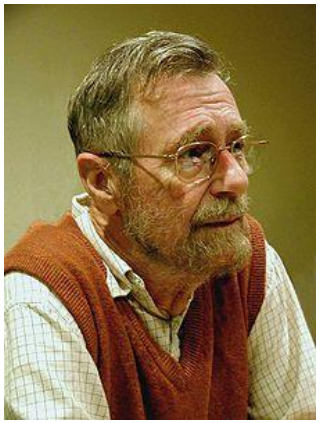
Idea: for all *v*, maintain upper bound d[*v*] on δ(s,v)

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```





Edsger Dijkstra



"Computer Science is no more about computers than astronomy is about telescopes."

- May 11, 1930 – August 6, 2002

- Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science.

- The Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000

- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

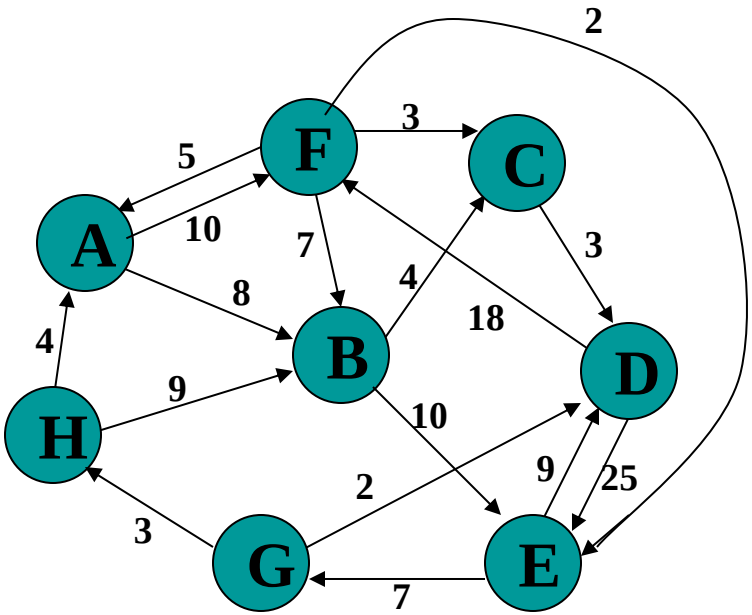
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's Algorithm - SSSP-Dijkstra

Dijkstra(G, w, s)
    InitializeSingleSource(G, s)
    S $\leftarrow \varnothing$
    Q $\leftarrow$ V[G]
    while Q $\neq$ 0 **do**
        u $\leftarrow$ *ExtractMin*(Q)
        S $\leftarrow$ S $\cup$ {u}
        for v $\in$ *Adj*[u] **do**
            Relax(u,v,w)

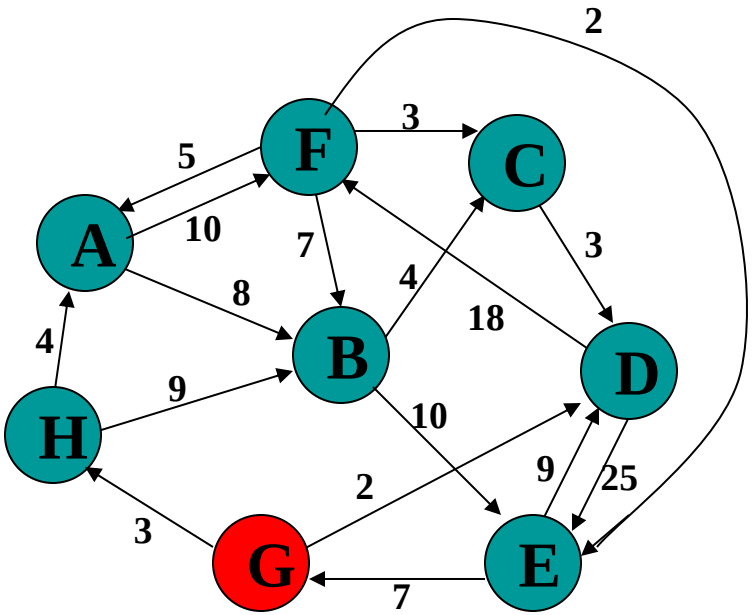
InitializeSingleSource(G, s)
    for v $\in$ V[G] **do**
        d[v] $\leftarrow \infty$
        p[v] $\leftarrow$ 0
    d[s] $\leftarrow$ 0

Example



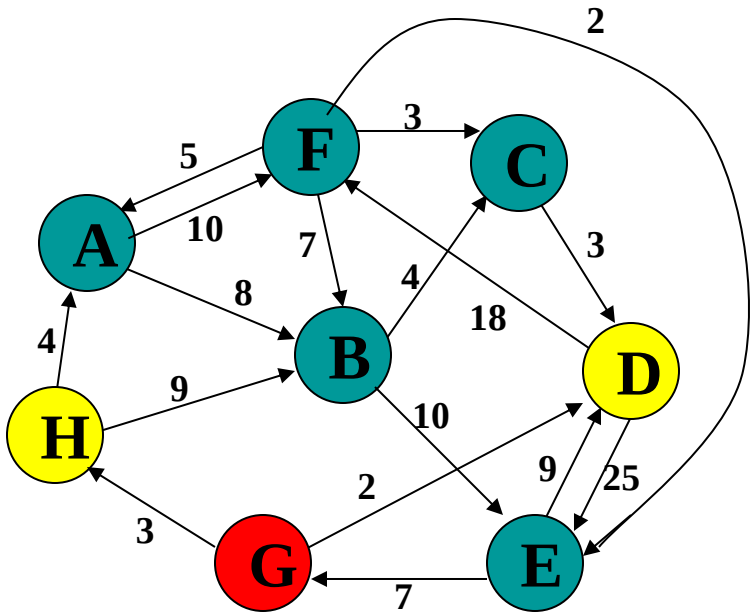
Initialize array

	S	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–



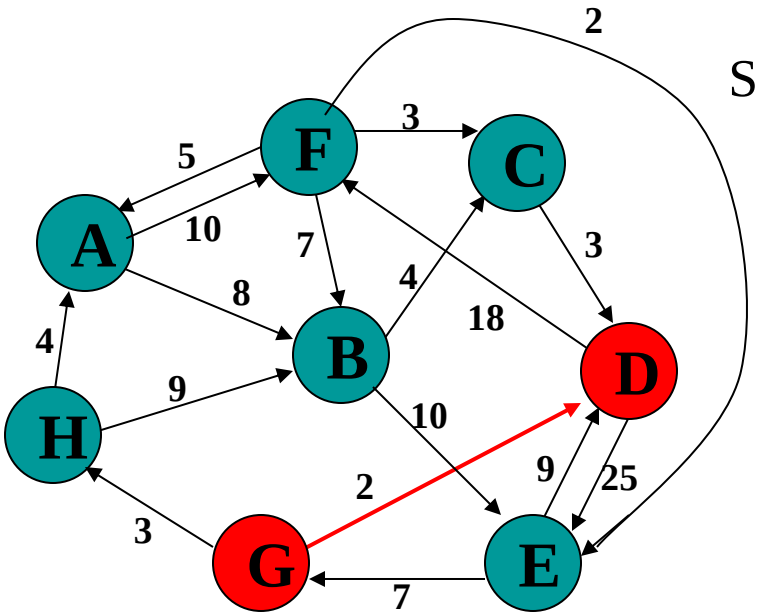
Start with G

	S	d_v	p_v
A			
B			
C			
D			
E			
F			
G	T	0	–
H			



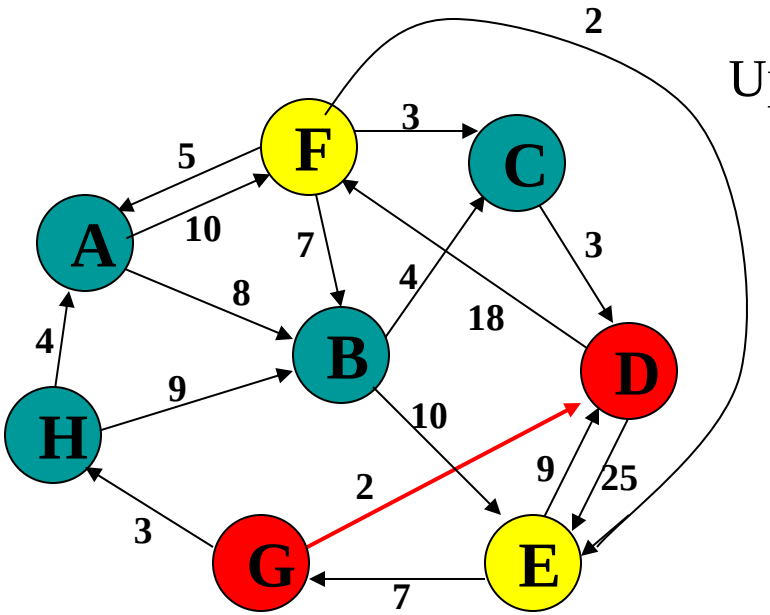
Update unselected nodes

	S	d_v	p_v
A			
B			
C			
D		2	G
E			
F			
G	T	0	–
H		3	G



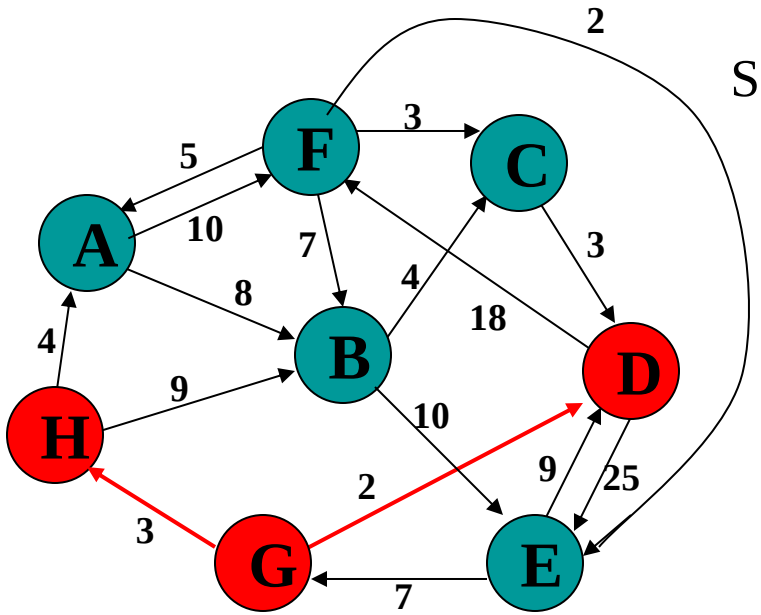
Select minimum distance

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E			
F			
G	T	0	–
H		3	G



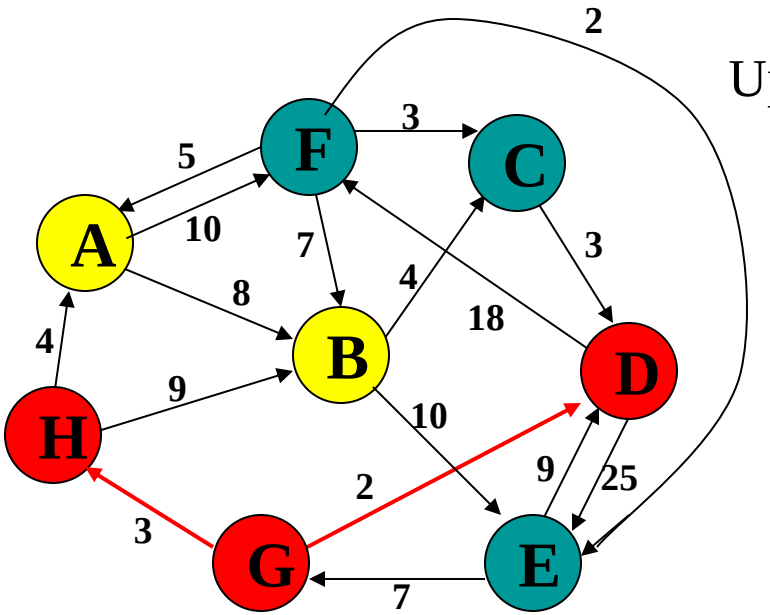
Update unselected nodes

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H		3	G



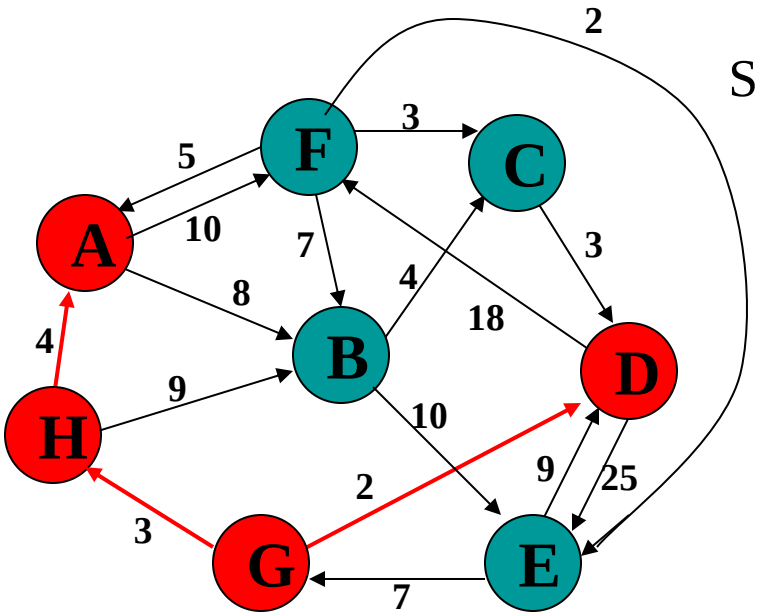
Select minimum distance

	S	d_v	p_v
A			
B			
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



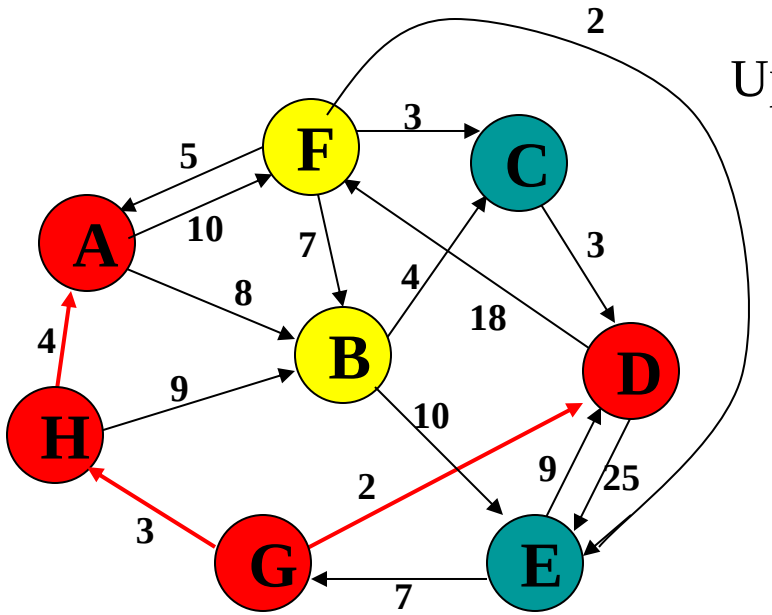
Update unselected nodes

	S	d_v	p_v
A		7	H
B		12	H
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



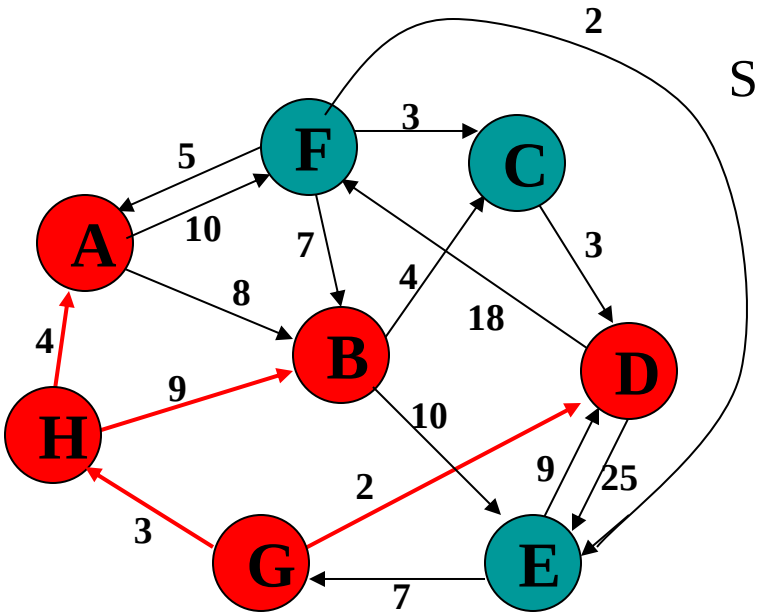
Select minimum distance

	S	d_v	p_v
A	T	7	H
B		12	H
C			
D	T	2	G
E		27	D
F		20	D
G	T	0	–
H	T	3	G



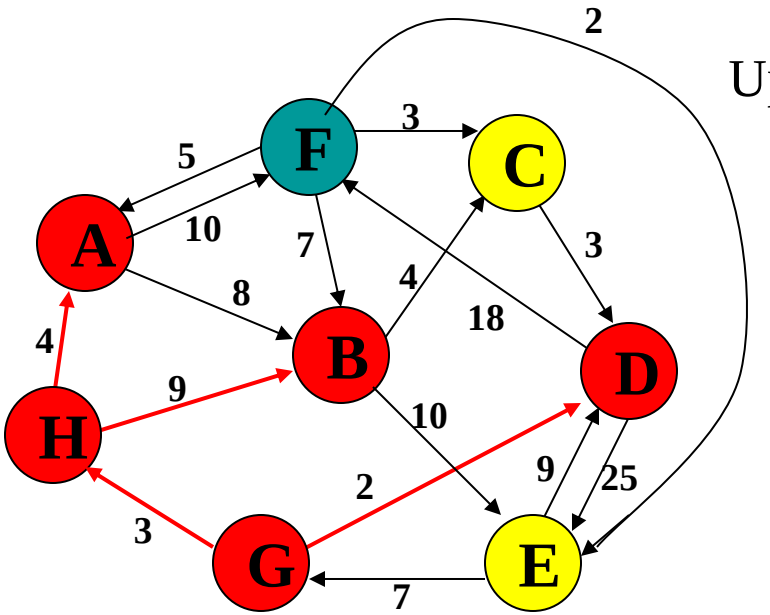
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B		12	H
C			
D	T	2	G
E		27	D
F		17	A
G	T	0	–
H	T	3	G



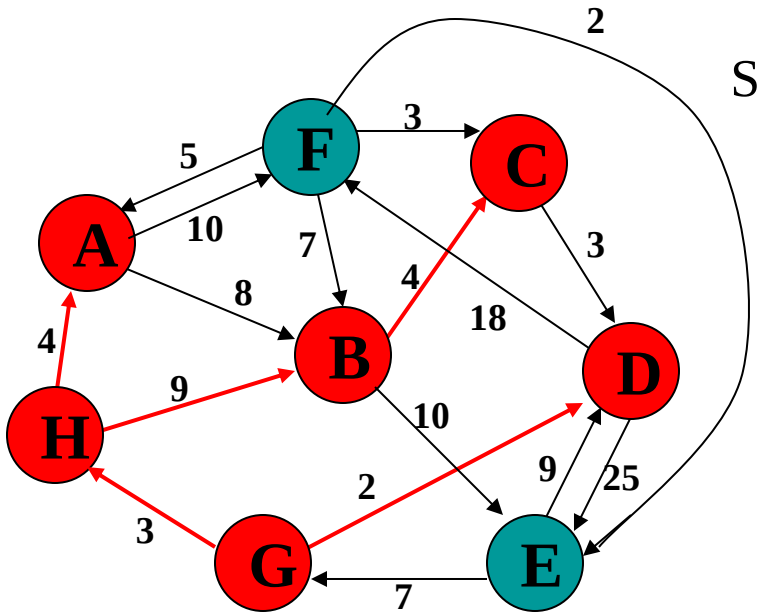
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C			
D	T	2	G
E		27	D
F		17	A
G	T	0	–
H	T	3	G



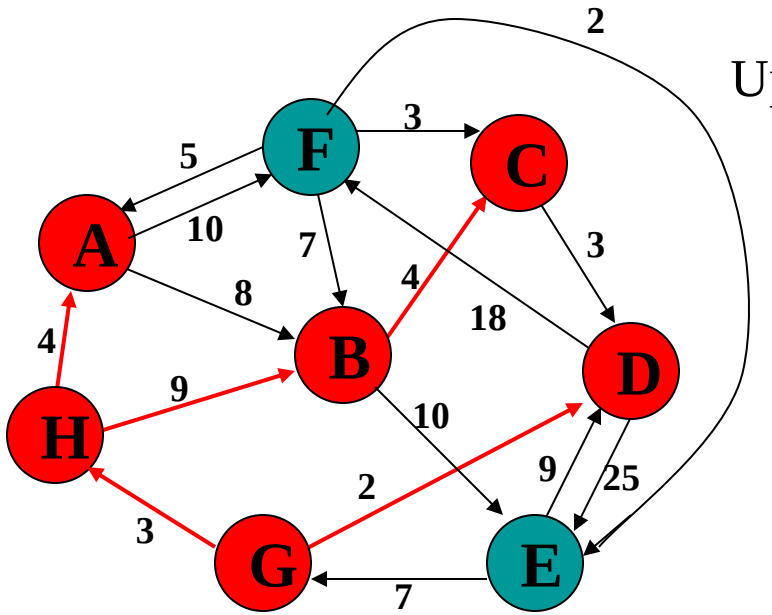
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C		16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



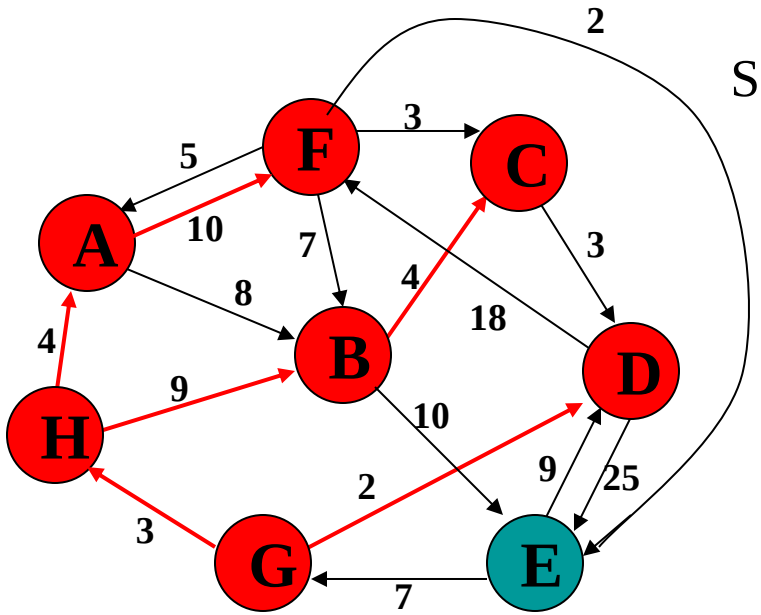
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



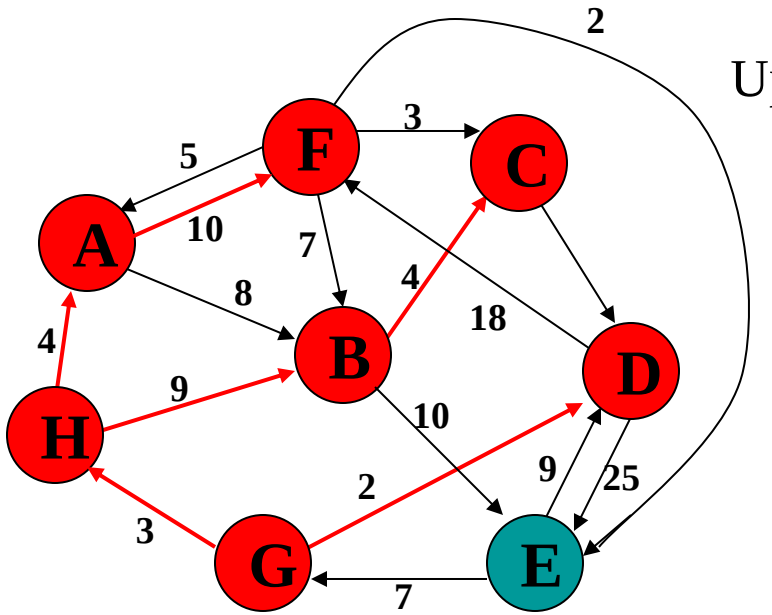
Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F		17	A
G	T	0	–
H	T	3	G



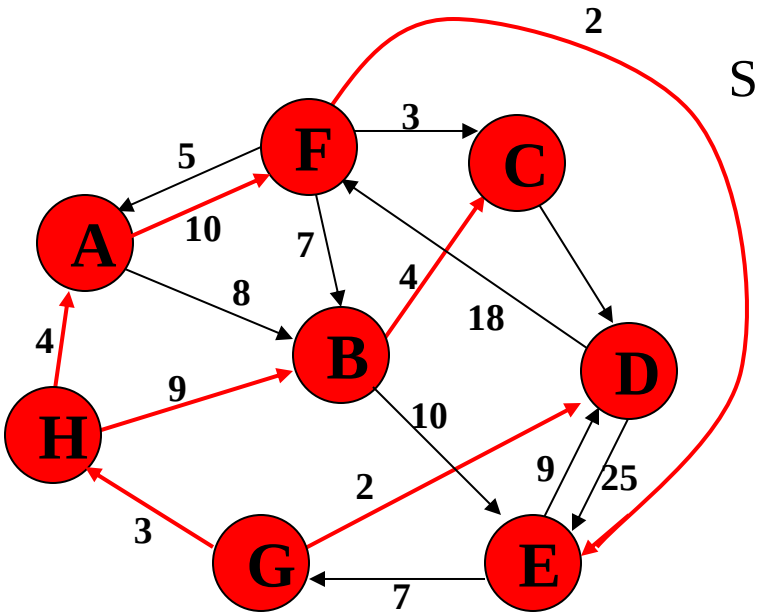
Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		22	B
F	T	17	A
G	T	0	–
H	T	3	G



Update unselected nodes

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E		19	F
F	T	17	A
G	T	0	–
H	T	3	G



Select minimum distance

	S	d_v	p_v
A	T	7	H
B	T	12	H
C	T	16	B
D	T	2	G
E	T	19	F
F	T	17	A
G	T	0	–
H	T	3	G

Done

Order of Complexity

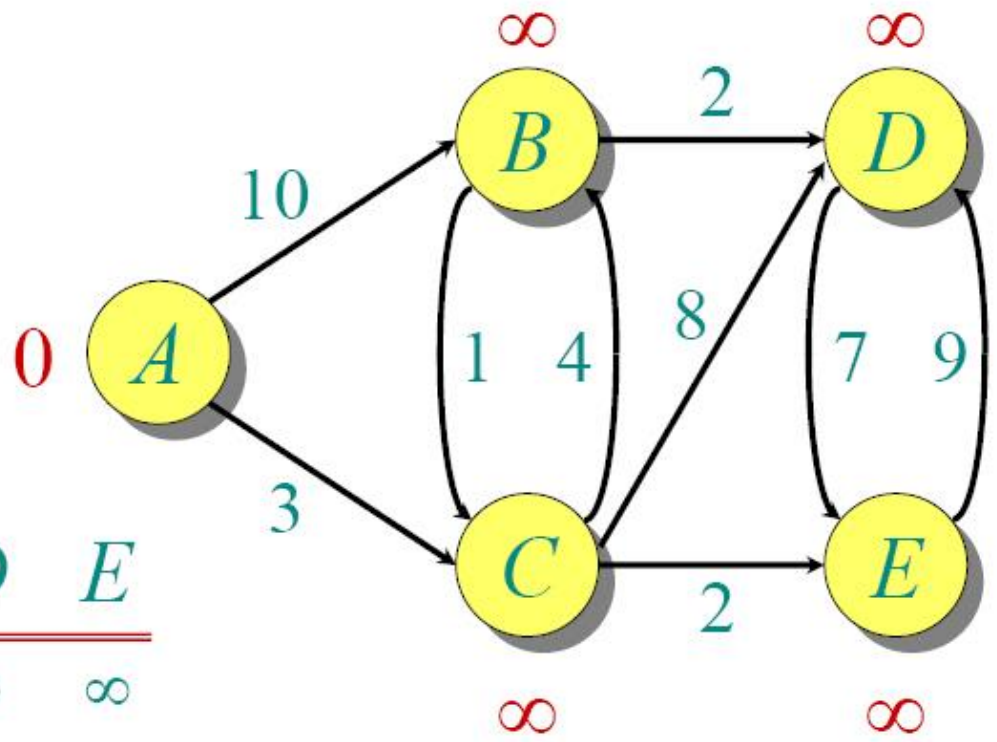
- Analysis
  - findMin() takes O(V) time
  - outer loop iterates (V-1) times
  - ➔ $O(V^2)$ time
- Optimal for dense graphs, i.e., $|E| = O(V^2)$
- Suboptimal for sparse graphs, i.e., $|E| = O(V)$

All-Pairs Shortest Paths

- One option: run Dijktra's algorithm |V| times ➔ $O(V^3)$ time
- There is a more efficient $O(V^3)$ time algorithm

Dijkstra Another Example

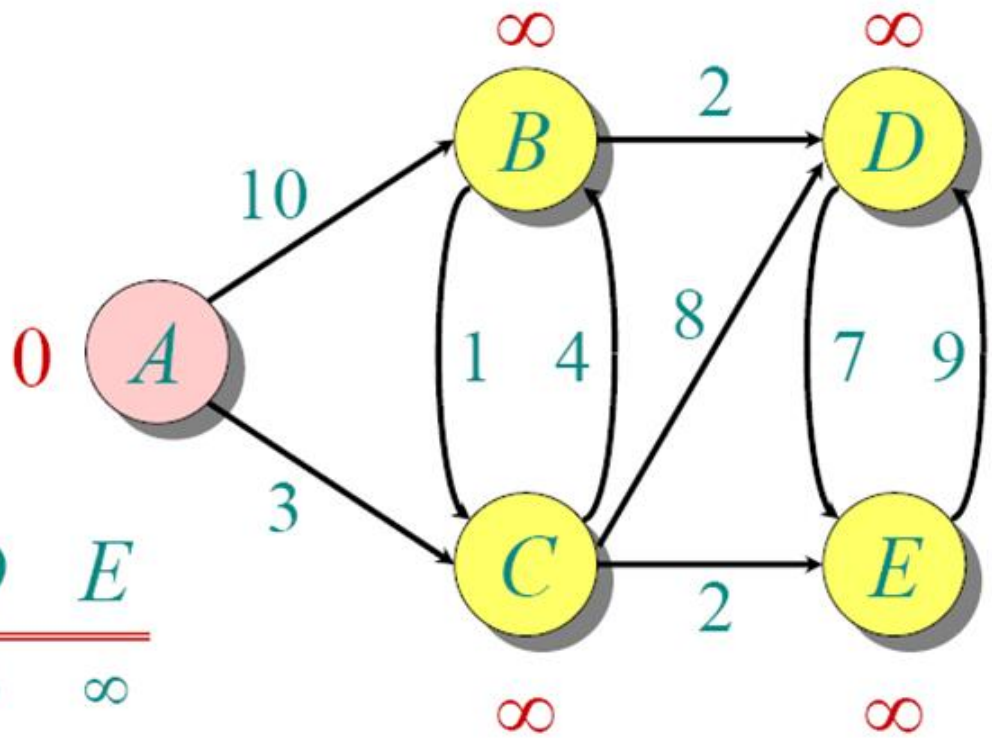
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

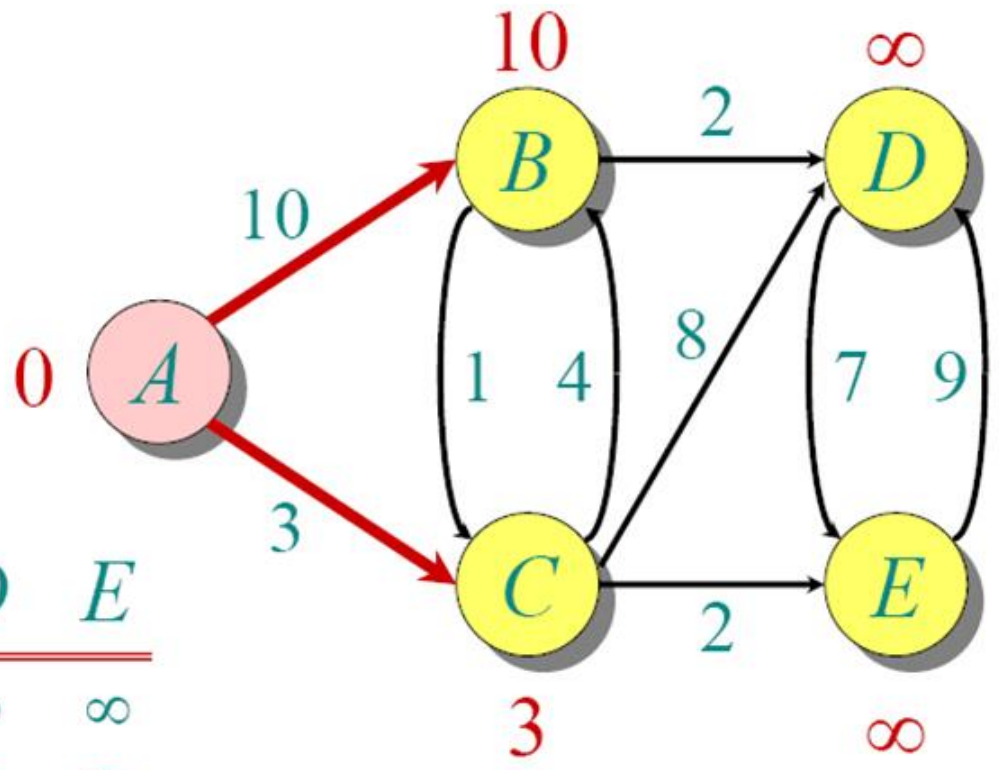
S: {}

Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

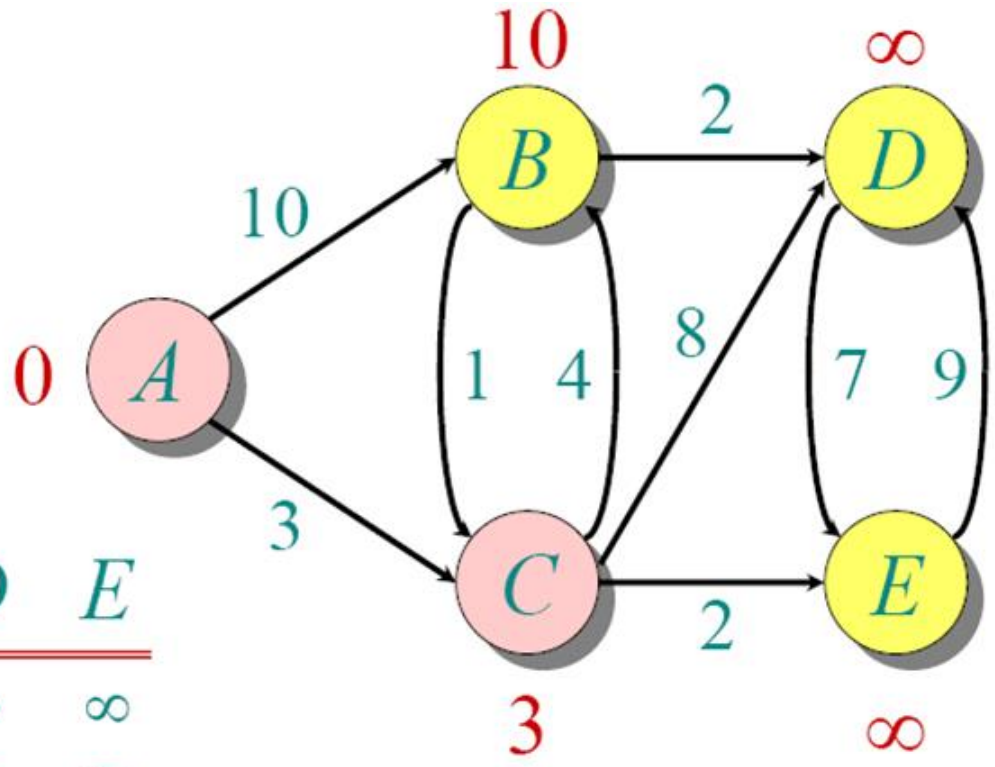
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

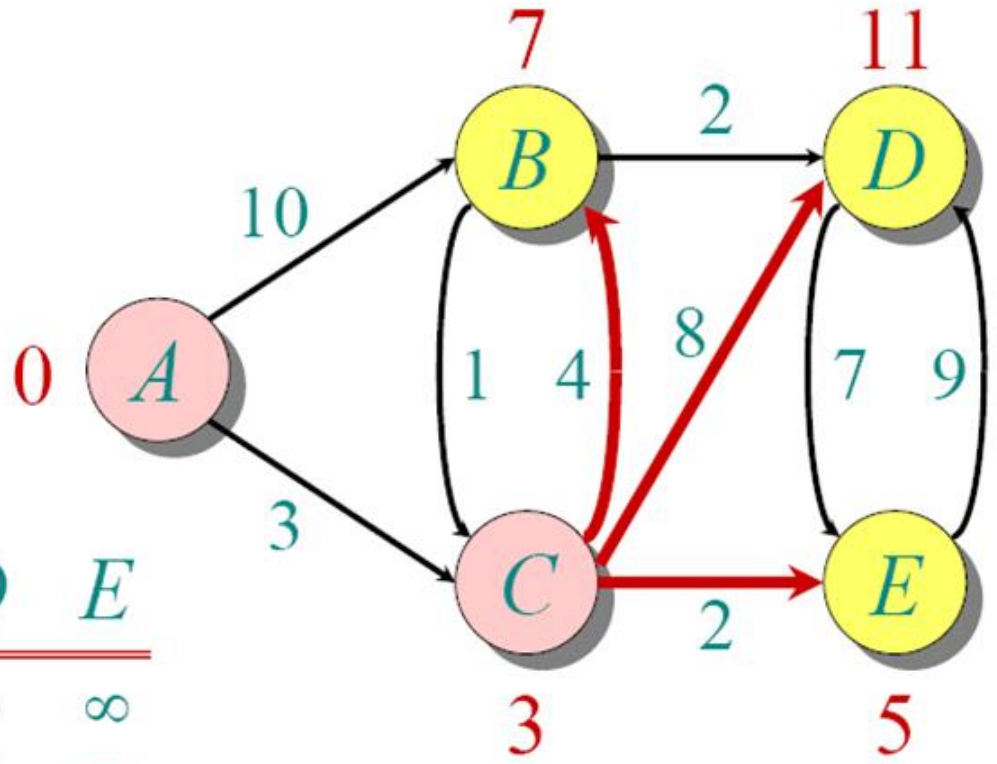
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

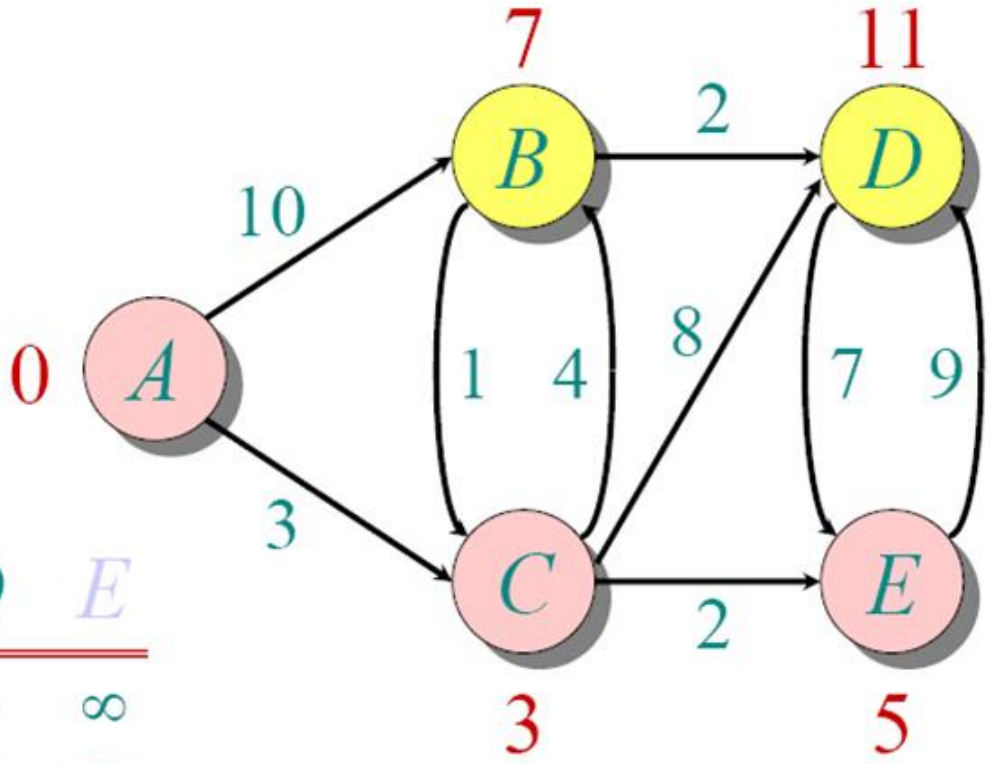
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C \}$

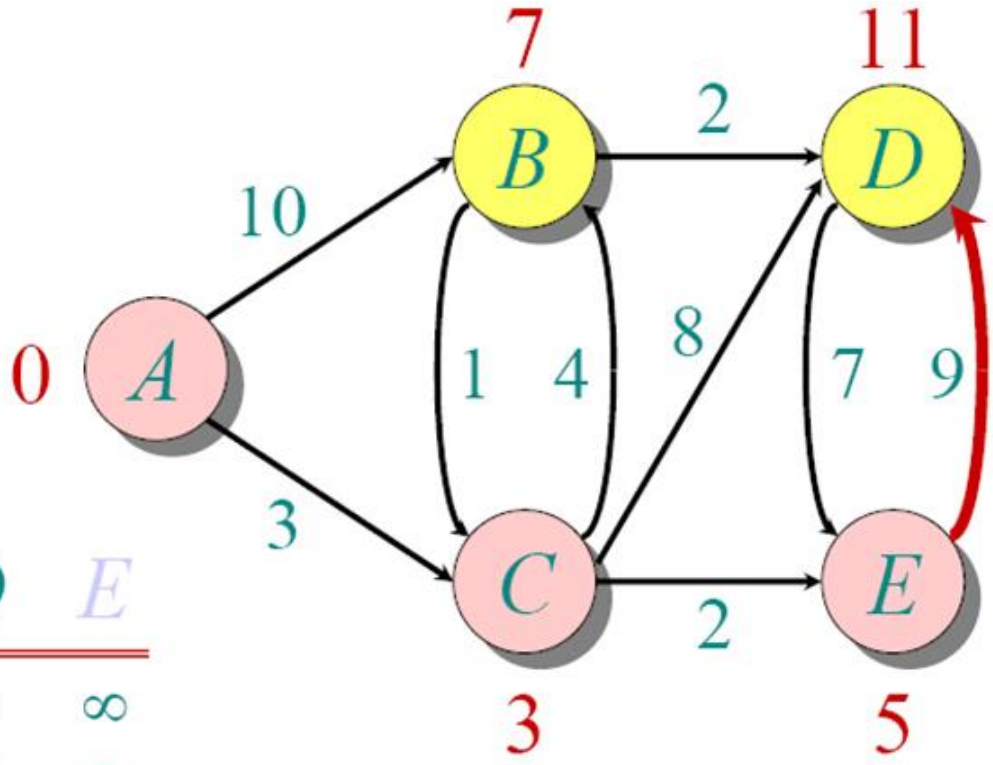
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C, E \}$

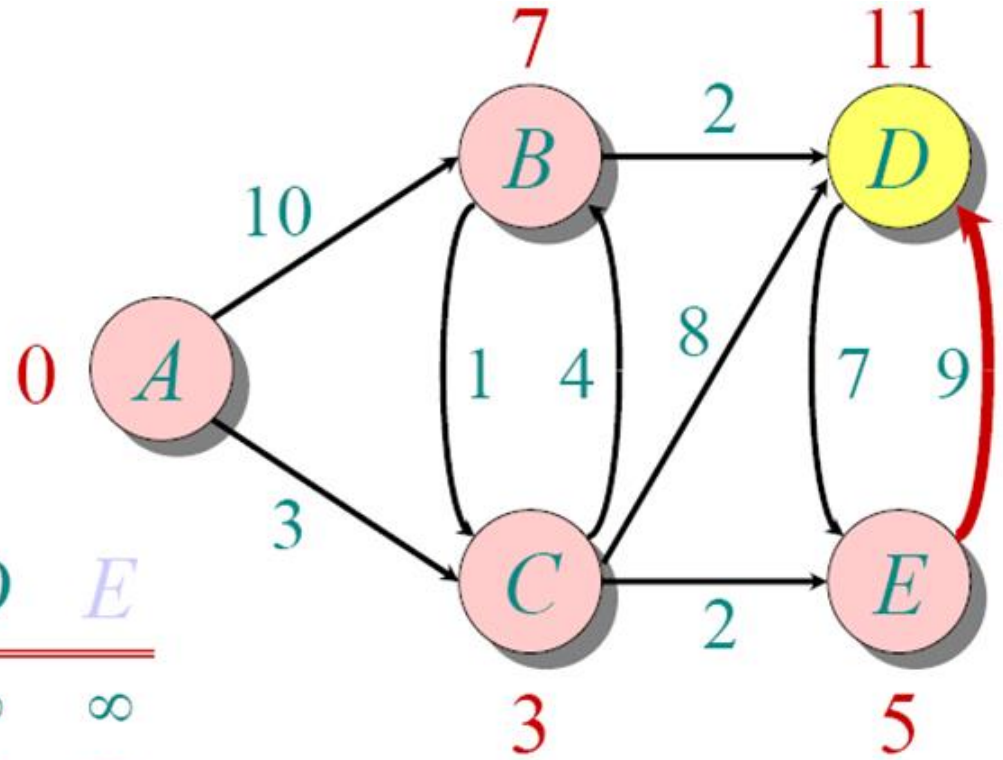
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

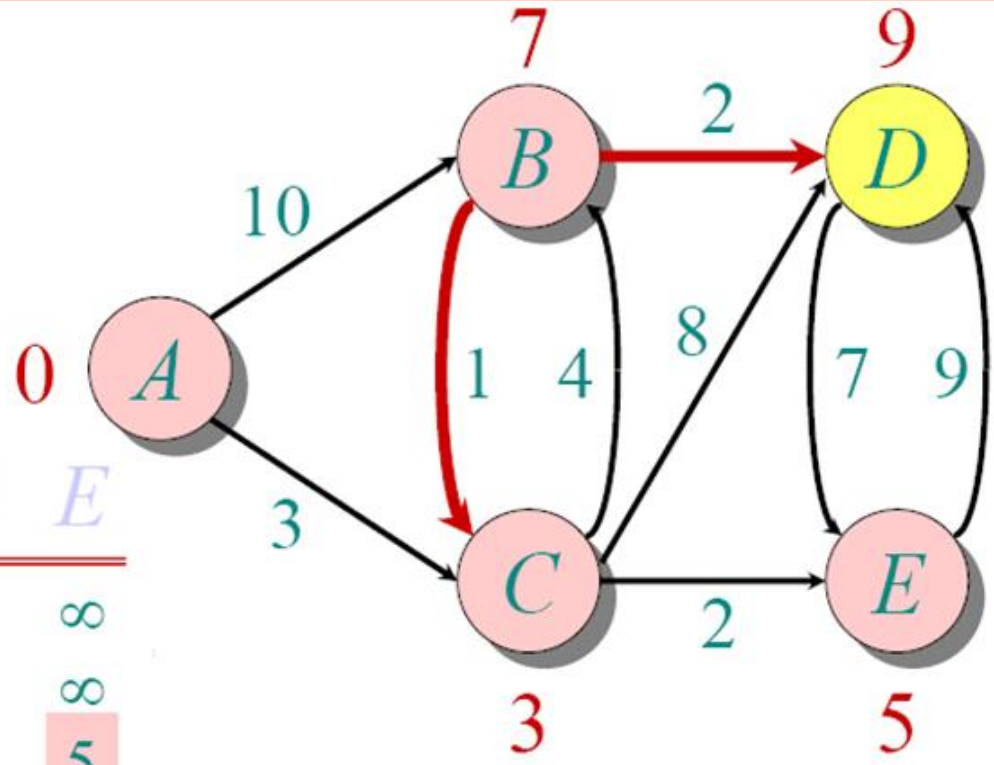
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

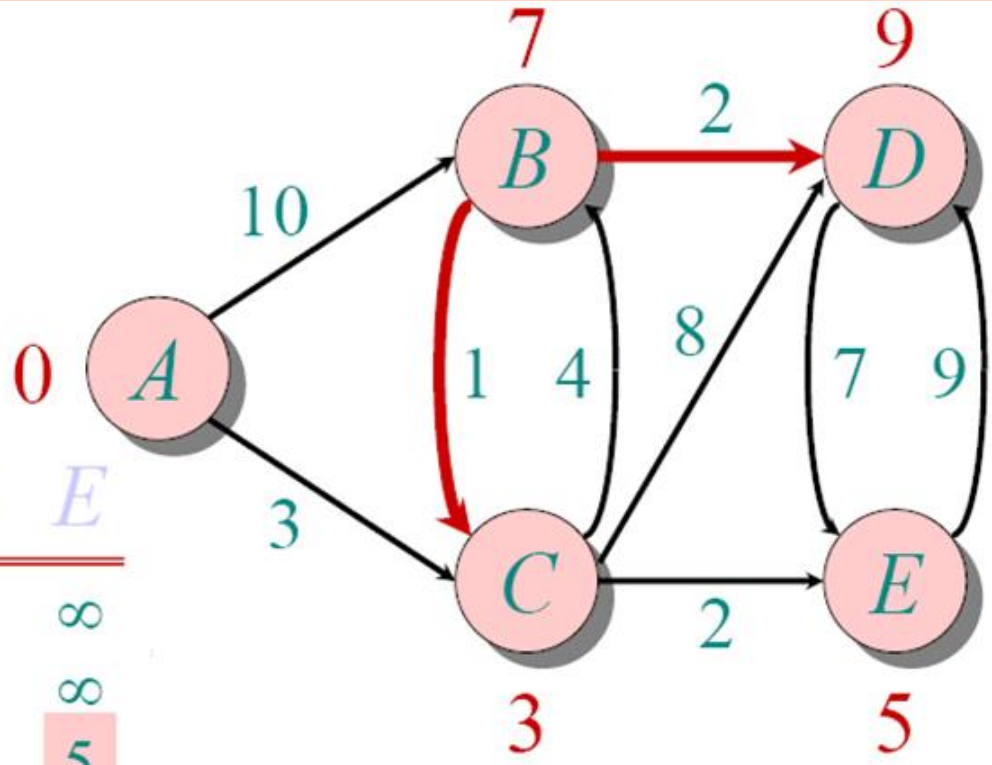
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

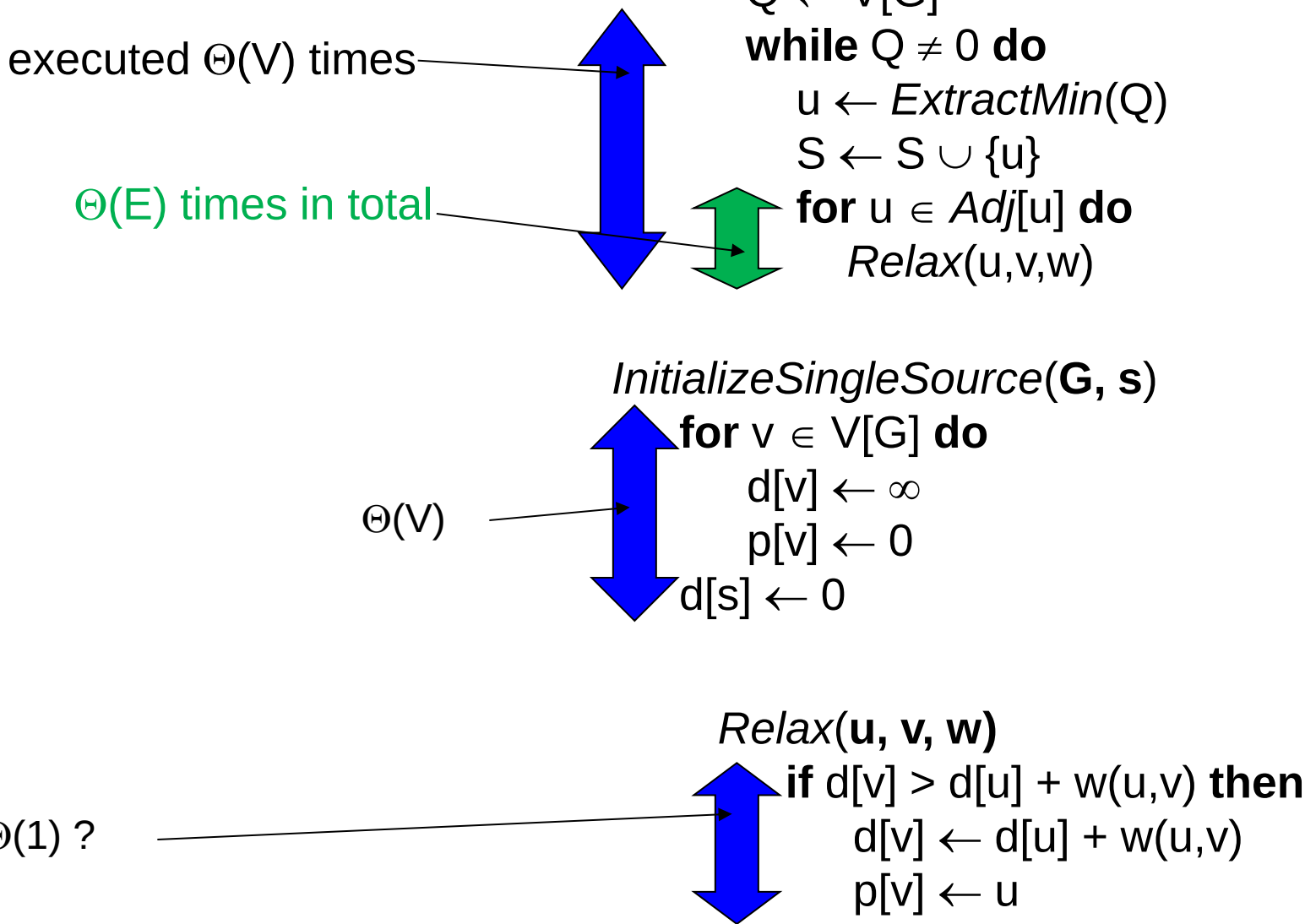
Dijkstra Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Dijkstra's - Complexity



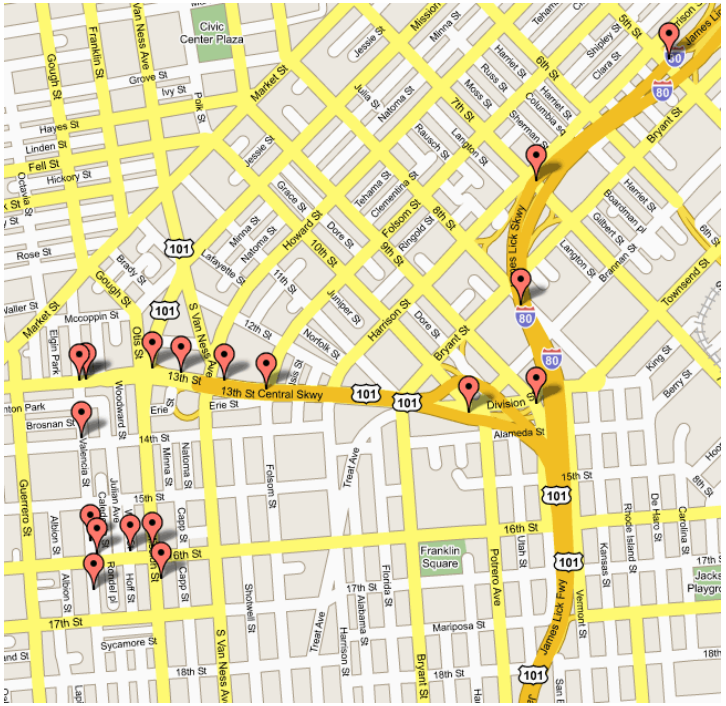
Dijkstra's Running Time

- Extract-Min executed $|V|$ time
- Decrease-Key (Relax) executed $|E|$ time
- Time = $|V| \, T_{\text{Extract-Min}} + |E| \, T_{\text{Decrease-Key}}$
- T depends on different Q implementations

Prority Queue	Extract-Min	Decrease-Key	Total
array	O(V)	O(1)	$O(V^2)$
binary heap	O(lg V)	O(lg V)	O(E lg V)
Fibonacci heap	O(lg V)	O(1) (amort.)	O(V lgV + E)

Applications of Dijkstra's Algorithm

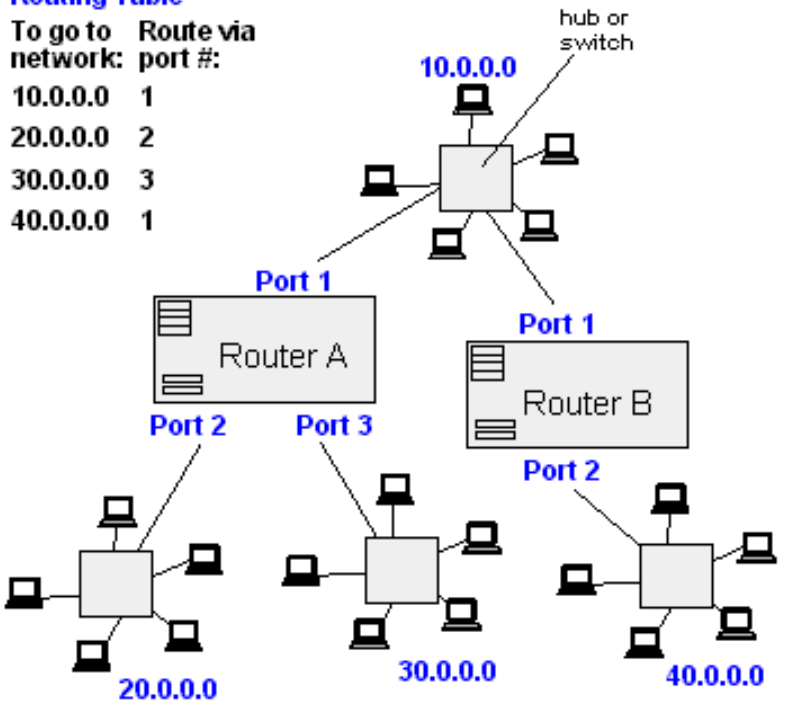
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Router A
Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1

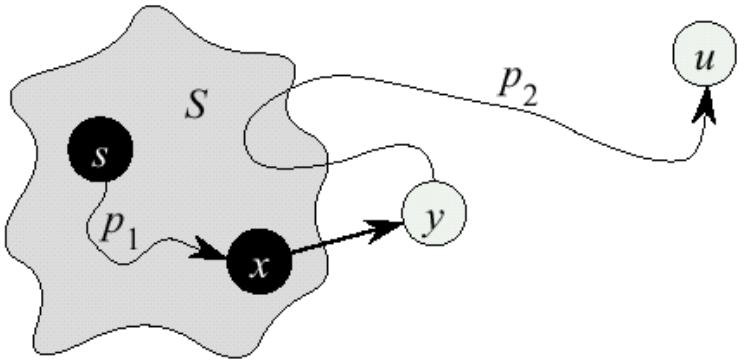


Dijkstra's Algorithm - Summary

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use *Q*, priority queue keyed by *d*[*v*] (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some *d* decreases)
- Basic idea
  - maintain a set *S* of solved vertices
  - at each step select "closest" vertex *u*, add it to *S*, and relax all edges from *u*

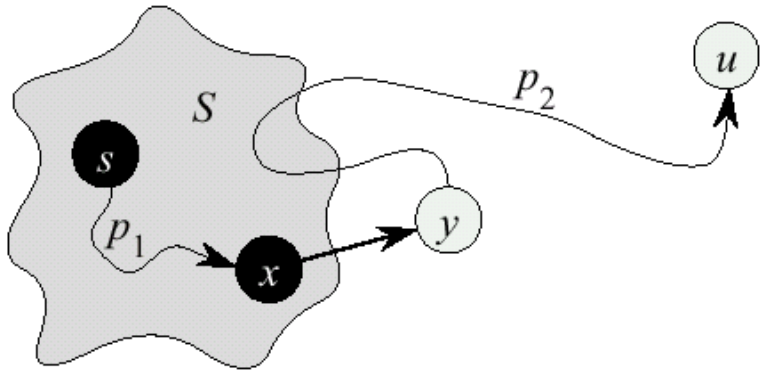
Dijkstra's Correctness

- We will prove that **whenever *u* is added to *S***, $d[u] = \delta(s,u)$, i.e., that d is minimum, and that equality is maintained thereafter
- Proof
  - Note that $\forall v$, $d[v] \geq \delta(s,v)$
  - Let u be the first **vertex picked** such that there is a shorter path than $d[u]$, i.e., that $\Rightarrow d[u] > \delta(s,u)$
  - We will show that this assumption leads to a contradiction



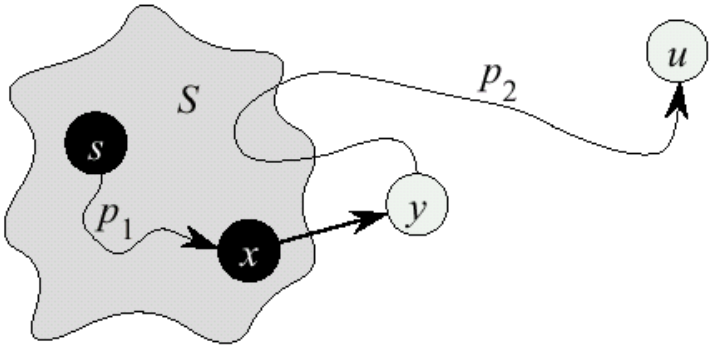
Dijkstra Correctness (2)

- Let y be the first vertex $\in V - S$ on the actual shortest path from s to u, then it must be that $d[y] = \delta(s,y)$ because
  - $d[x]$ is set correctly for y's predecessor $x \in S$ on the shortest path (by choice of u as the first vertex for which d is set incorrectly)
  - when the algorithm inserted x into S, it relaxed the edge (x,y), assigning $d[y]$ the correct value



Dijkstra Correctness (3)

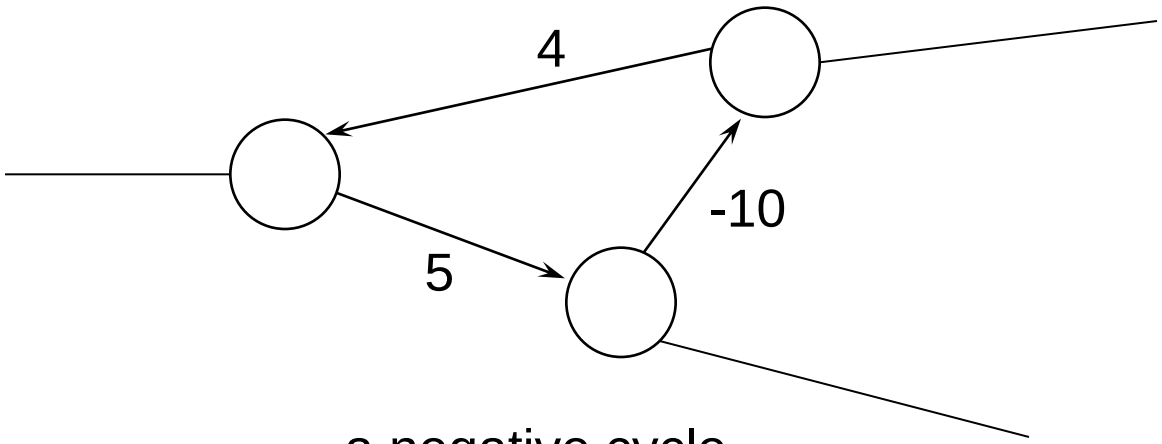
$$
\begin{aligned}
d[u] &> \delta(s,u) && \text{(initial assumption)} \\
&= \delta(s,y)+\delta(y,u) && \text{(optimal substructure)} \\
&= d[y]+\delta(y,u) && \text{(correctness of } d[y]\text{)} \\
&\geq d[y] && \text{(no negative weights)}
\end{aligned}
$$



- But $d[u] > d[y] \Rightarrow$ algorithm would have chosen y (from the PQ) to process next, not $u \Rightarrow$ Contradiction
- Thus d[u] = $\delta(s,u)$ at time of insertion of u into S, and Dijkstra's algorithm is correct

The Bellman-Ford Algorithm

- Handles negative edge weights
- Detects negative cycles
- Is slower than Dijkstra



a negative cycle

Bellman-Ford: Idea

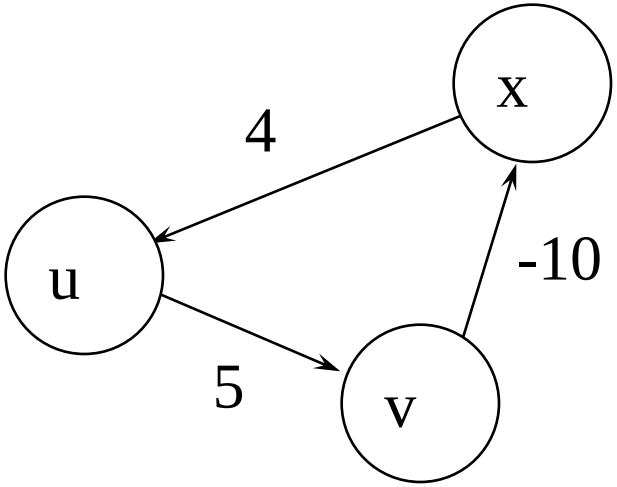
- Repeatedly update d for all pairs of vertices connected by an edge.
- **Theorem:** If *u* and *v* are two vertices with an edge from *u* to *v*, and $s \Rightarrow u \rightarrow v$ is a shortest path, and $d[u] = \delta(s,u)$,
- then $d[u]+w(u,v)$ is the length of a shortest path to *v*.
- **Proof:** Since $s \Rightarrow u \rightarrow v$ is a shortest path, its length is $\delta(s,u) + w(u,v) = d[u] + w(u,v)$. ■

Why Bellman-Ford Works

- On the first pass, we find δ(s,u) for all vertices whose shortest paths have one edge.
- On the second pass, the d[*u*] values computed for the one-edge-away vertices are correct (= δ(s,u)), so they are used to compute the correct d values for vertices whose shortest paths have two edges.
- Since no shortest path can have more than |V[G]|-1 edges, after that many passes all d values are correct.
- Note: all vertices not reachable from s will have their original values of infinity. (Same, by the way, for Dijkstra).

Negative Cycle Detection

- What if there is a negative-weight cycle reachable from s?
- Assume: $d[u] \leq d[x]+4$
- $d[v] \leq d[u]+5$
- $d[x] \leq d[v]-10$
- Adding:
- $d[u]+d[v]+d[x] \leq d[x]+d[u]+d[v]-1$
- Because it's a cycle, vertices on left are same as those on right. Thus we get $0 \leq -1$; a contradiction.
  So for at least one edge (u,v),
- $d[v] > d[u] + w(u,v)$
- This is exactly what Bellman-Ford checks for.



Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

Initialize d[], which will converge to shortest-path value δ

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

Test for solution

Under what condition do we get a solution?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

What will be the running time?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

What will be the running time?

A: O(VE)

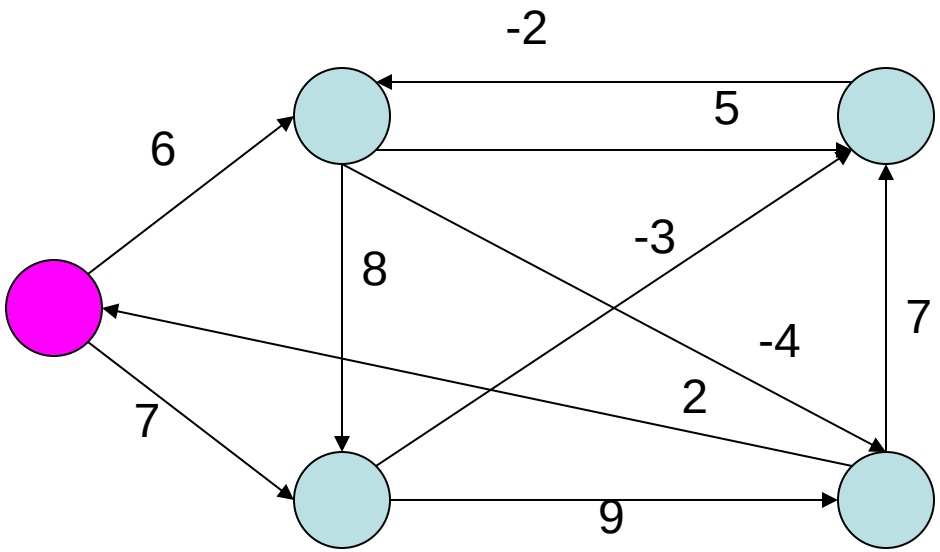
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

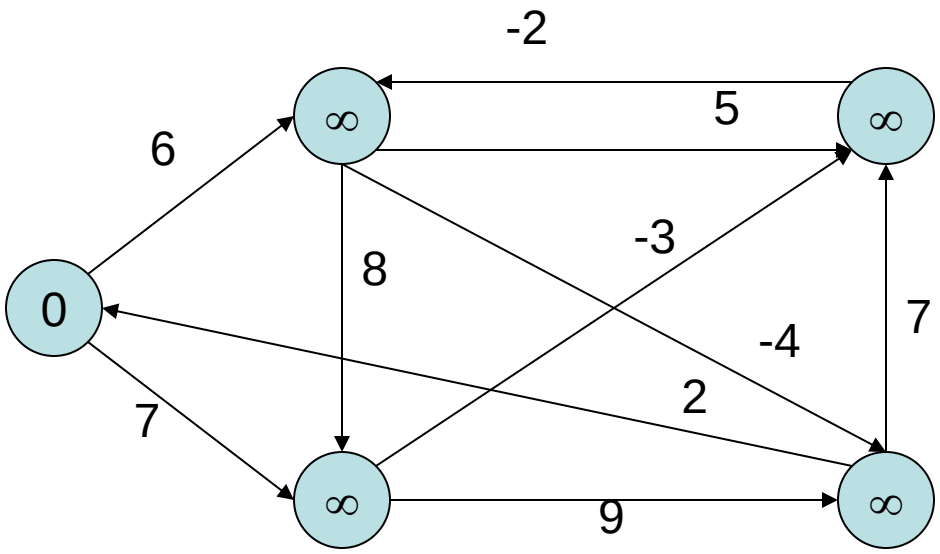
```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

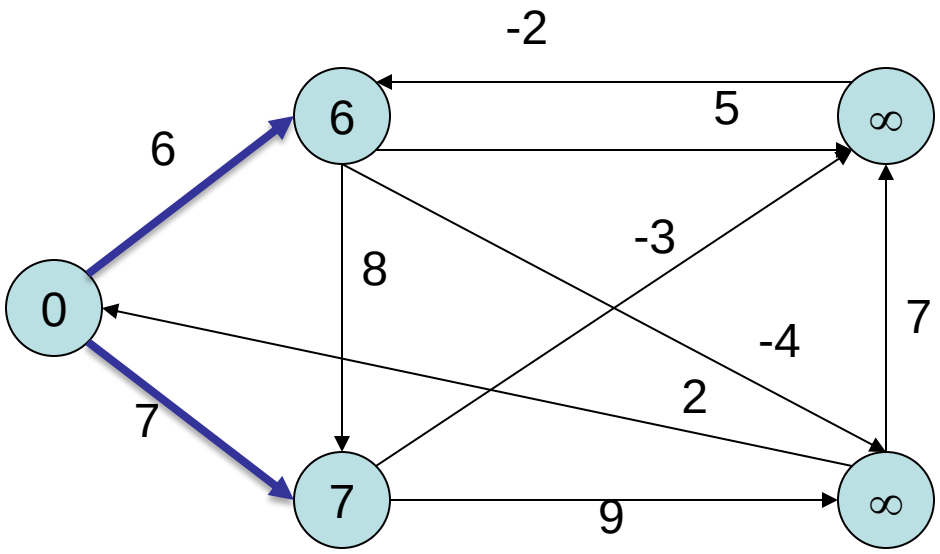
Bellman-Ford Algorithm - Example



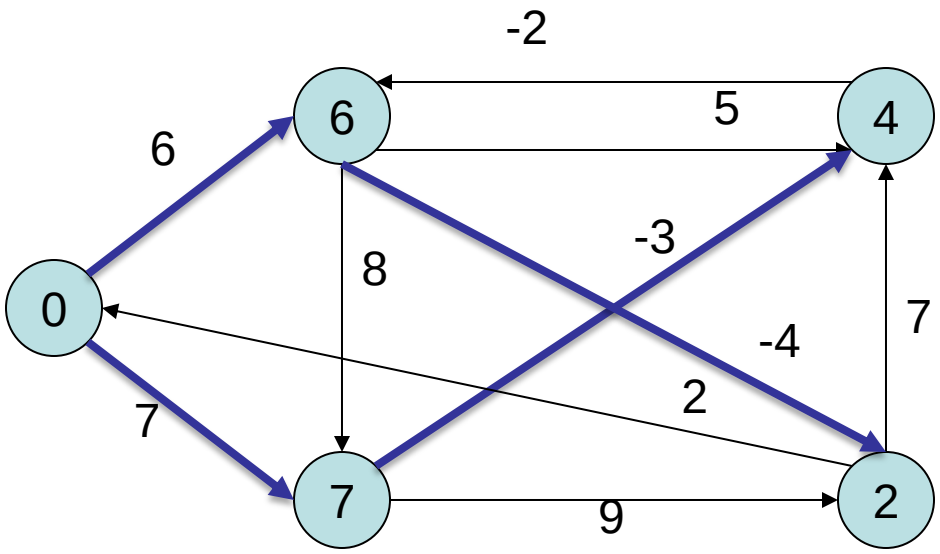
Bellman-Ford Algorithm - Example



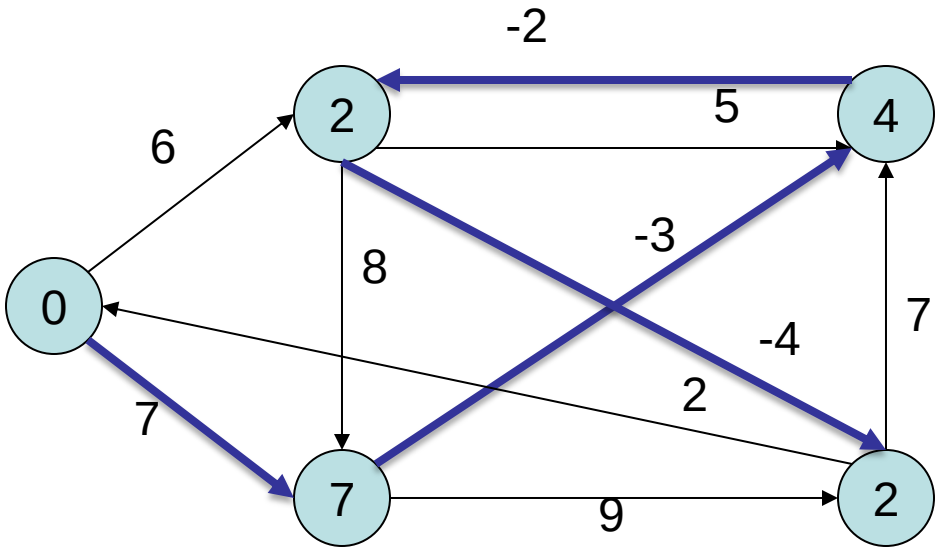
Bellman-Ford Algorithm - Example



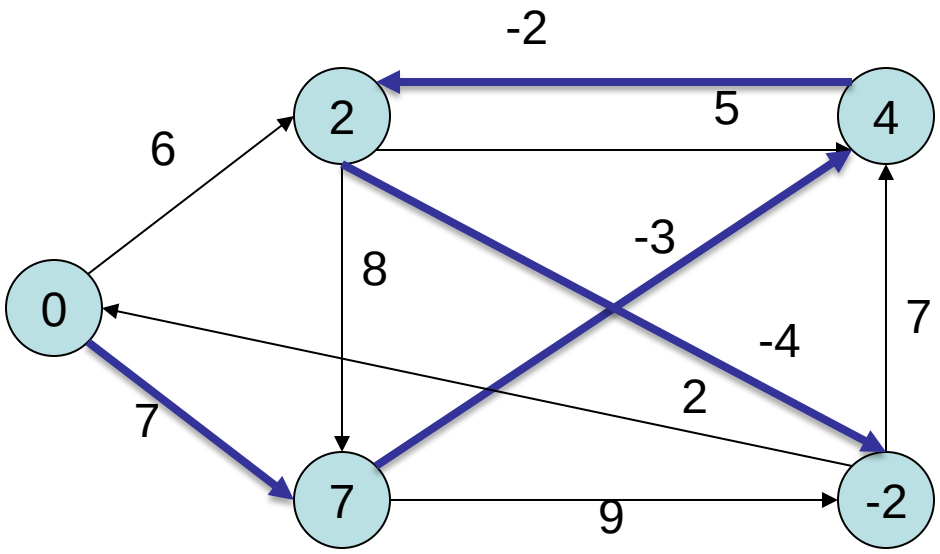
Bellman-Ford Algorithm - Example



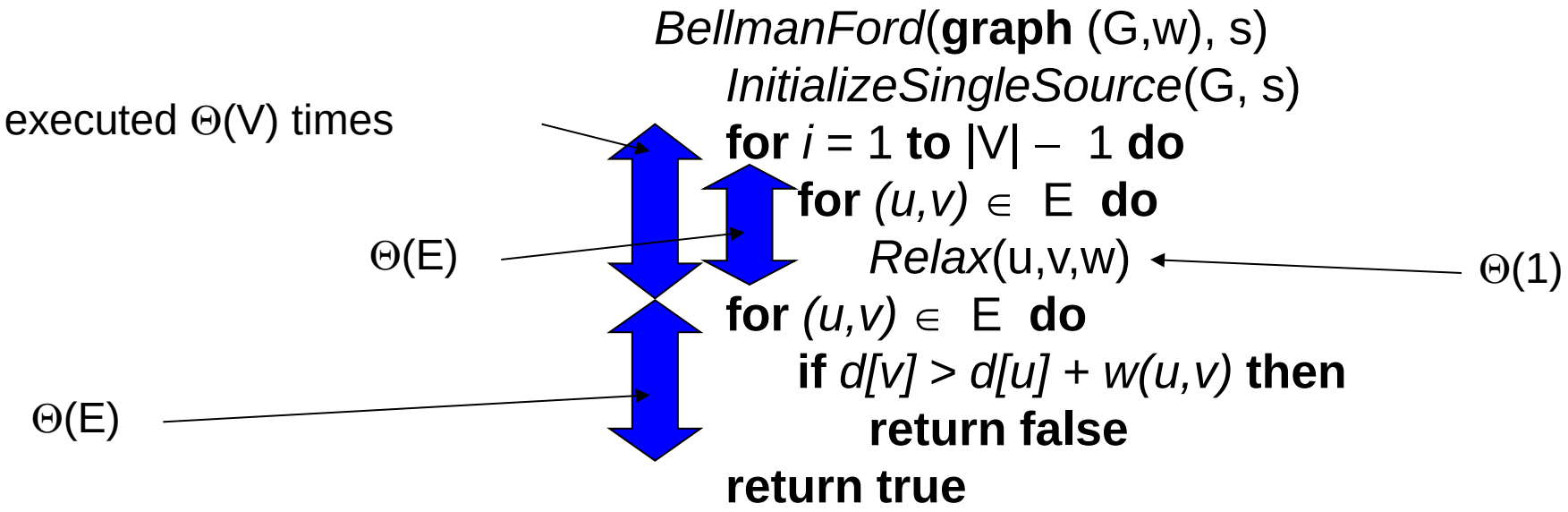
Bellman-Ford Algorithm - Example



Bellman-Ford Algorithm - Example

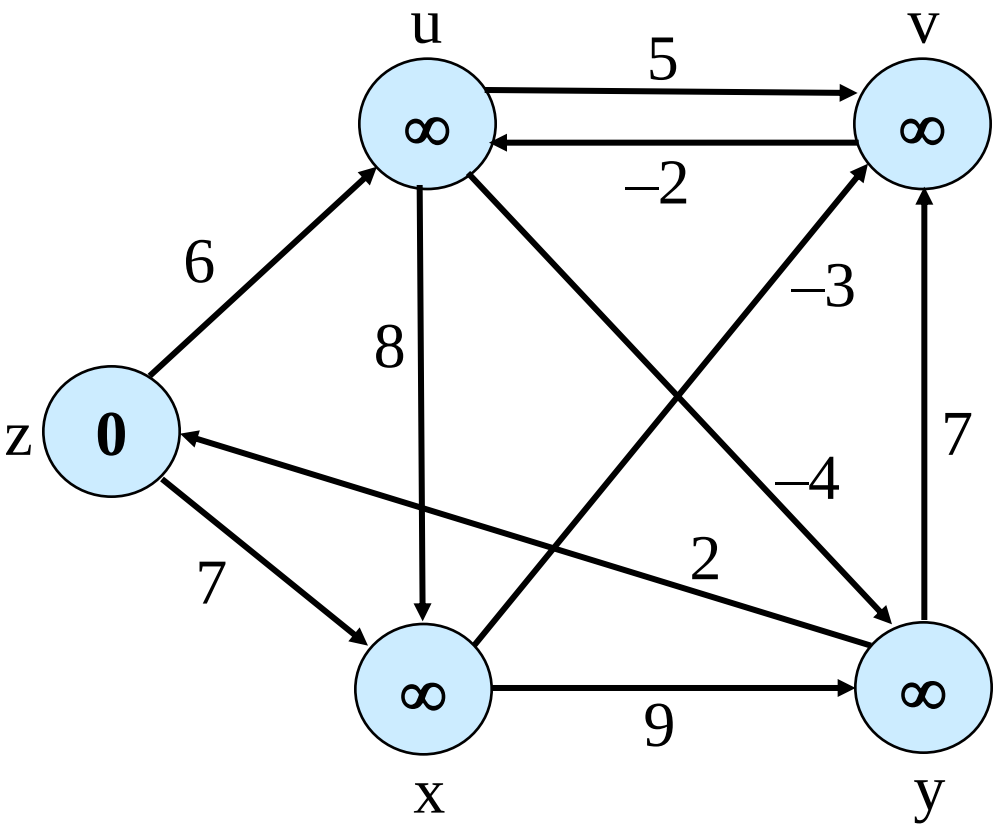


Bellman-Ford - Complexity



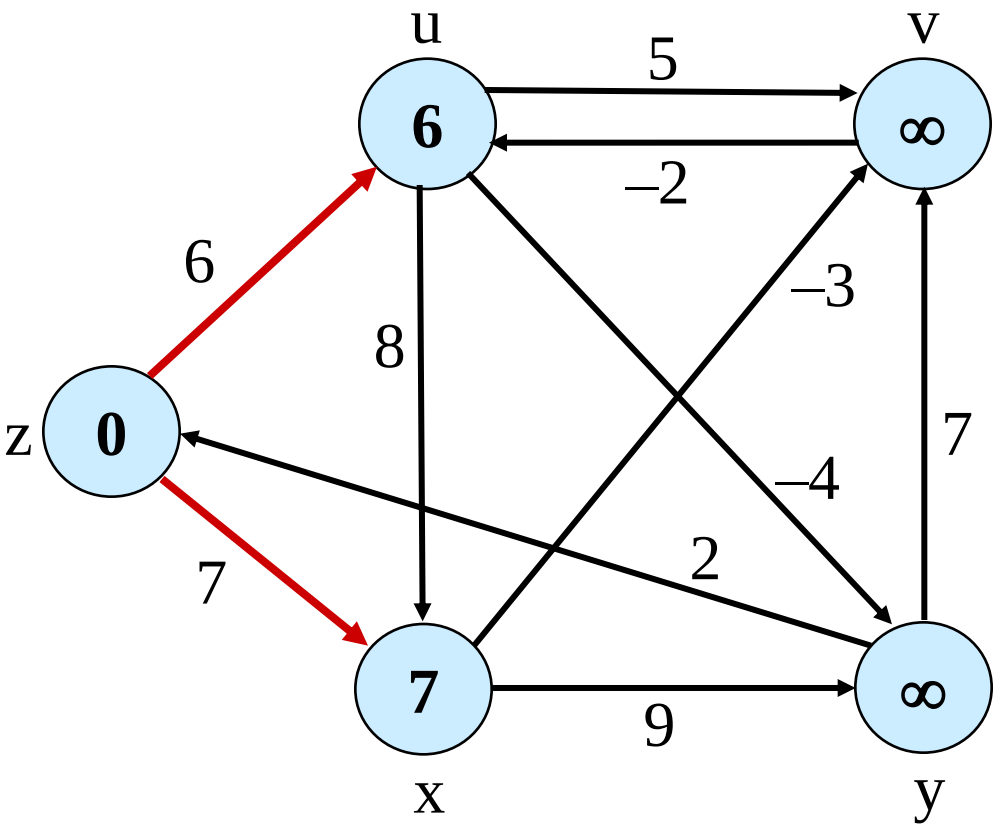
So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Example



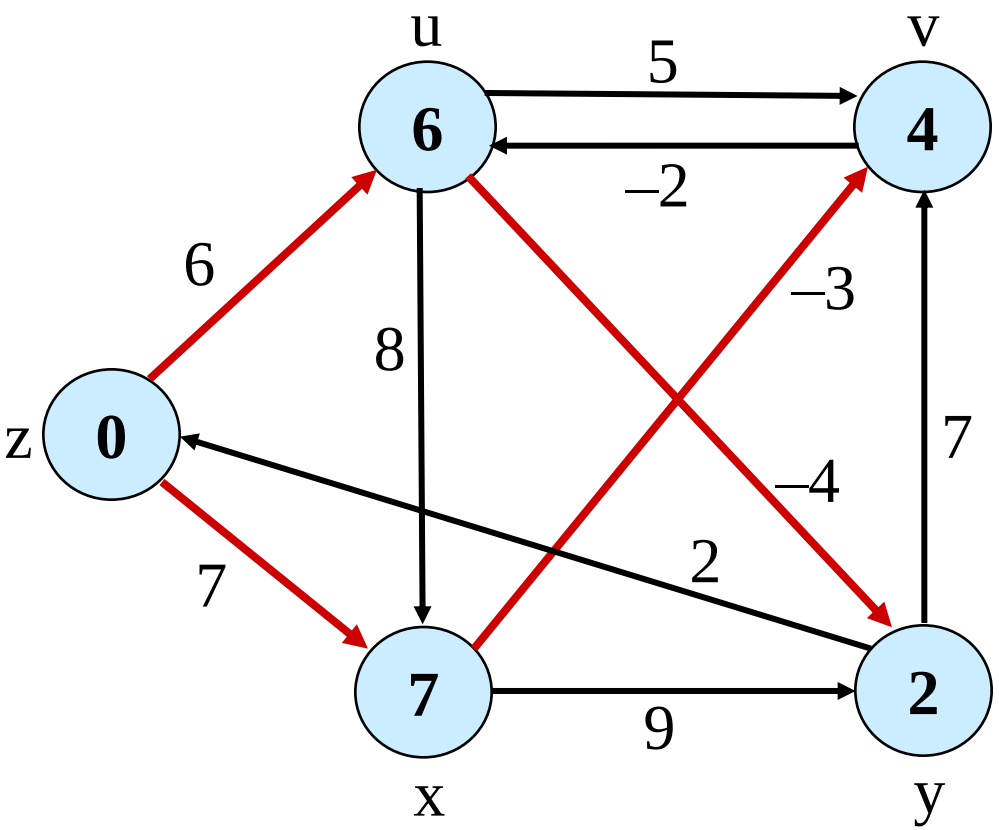
So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Example

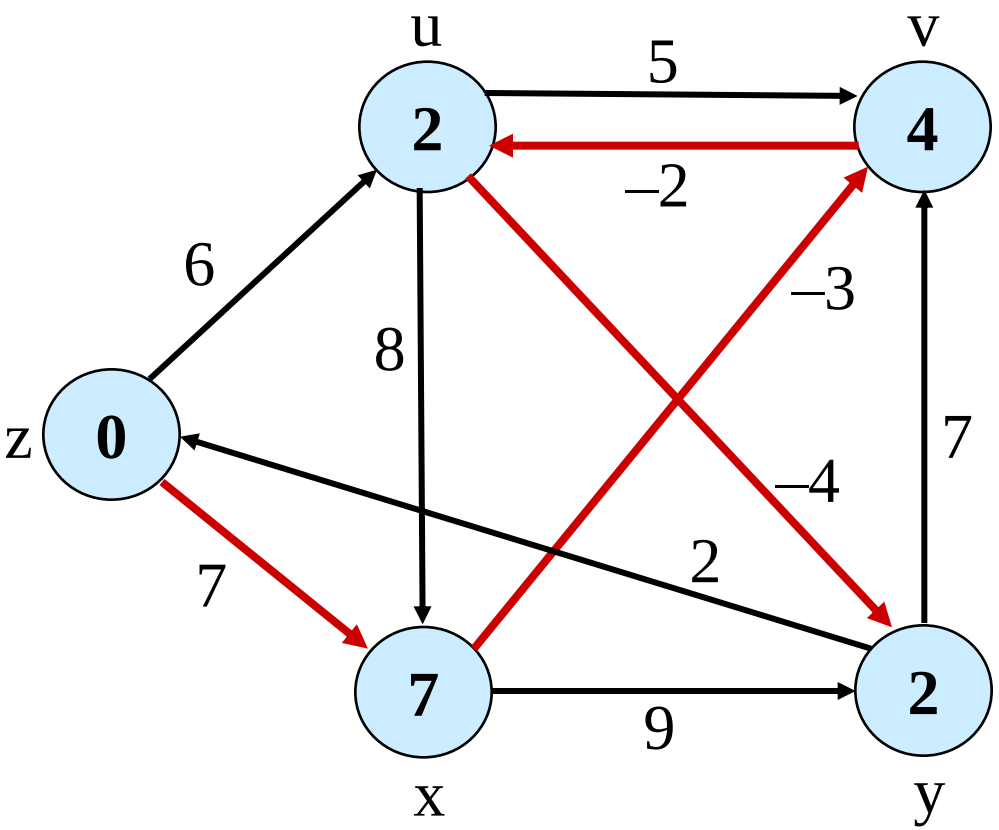


So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

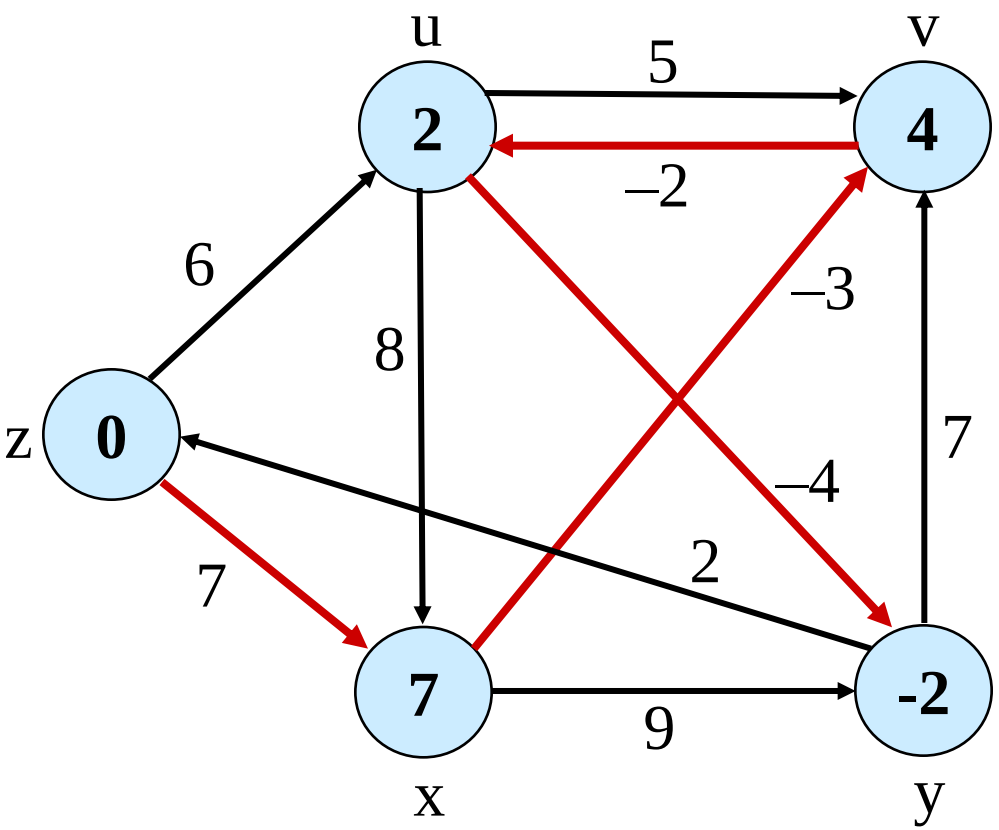
Example



Example



Example



Correctness of Bellman-Ford

- Let $\delta_i(s,u)$ denote the length of path from *s* to *u*, that is shortest among all paths, that contain at most *i* edges
- Prove by induction that $d[u] = \delta_i(s,u)$ after the *i*-th iteration of Bellman-Ford
  - Base case (i=0) trivial
  - Inductive step (say $d[u] = \delta_{i-1}(s,u)$):
    - Either $\delta_i(s,u) = \delta_{i-1}(s,u)$
    - Or $\delta_i(s,u) = \delta_{i-1}(s,z) + w(z,u)$
    - In an iteration we try to relax each edge ((z,u) also), so we will catch both cases, thus $d[u] = \delta_i(s,u)$

Correctness of Bellman-Ford

- After *n-1* iterations, $d[u] = \delta_{n-1}(s,u)$, for each vertex *u.*
- If there is still some edge to relax in the graph, then there is a vertex *u*, such that $\delta_n(s,u) < \delta_{n-1}(s,u)$. But there are only *n* vertices in *G* – we have a cycle, and it must be negative.
- Otherwise, $d[u] = \delta_{n-1}(s,u) = \delta(s,u)$, for all *u,* since any shortest path will have at most *n-1* edges

DAG Shortest Paths

- Problem: finding shortest paths in DAG
  - Bellman-Ford takes O(VE) time.
  - *How can we do better?*
  - Idea: use topological sort
    - If were lucky and processes vertices on each shortest path from left to right, would be done in one pass
    - Every path in a dag is subsequence of topologically sorted vertex order, so processing verts in that order, we will do each path in forward order (will never relax edges out of vert before doing all edges into vert).
    - Thus: just one pass. *What will be the running time?*

Shortest Paths in DAGs - SP-DAG

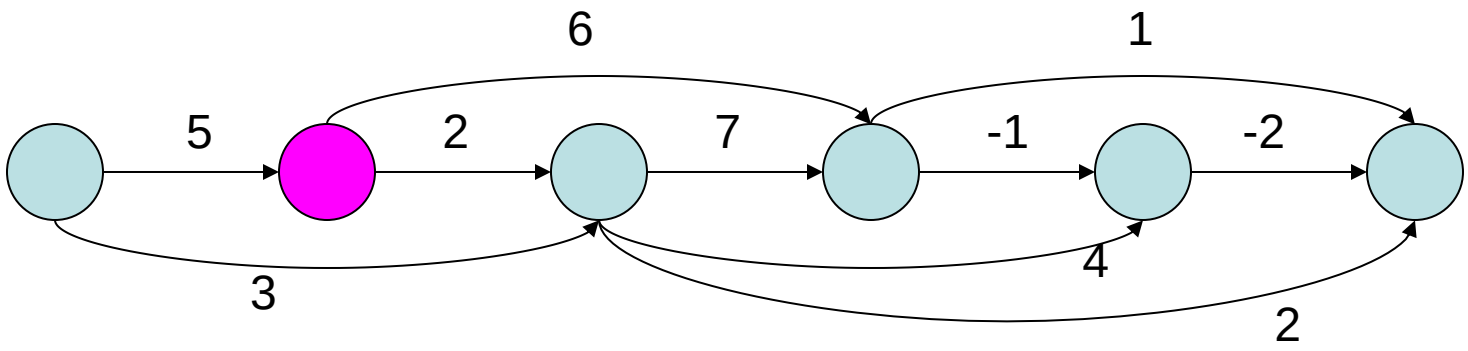
SP-DAG(**graph** (G,w), **vertex** s)

topologically sort vertices of G

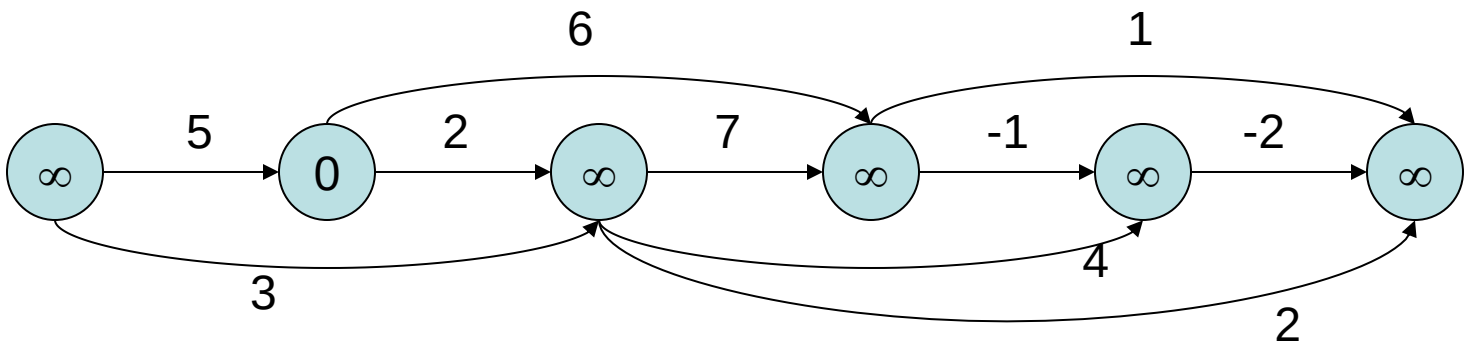
InitializeSingleSource(G, s)

for each vertex u taken in topologically sorted order **do**
    for each vertex $v \in$ *Adj*[u] **do**
        Relax(u,v,w)

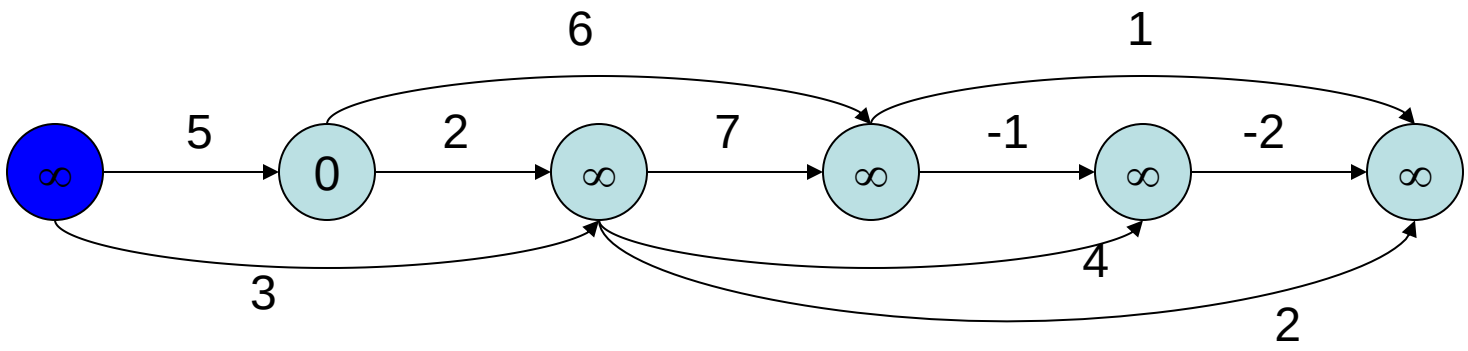
Shortest Paths in DAGs - Example



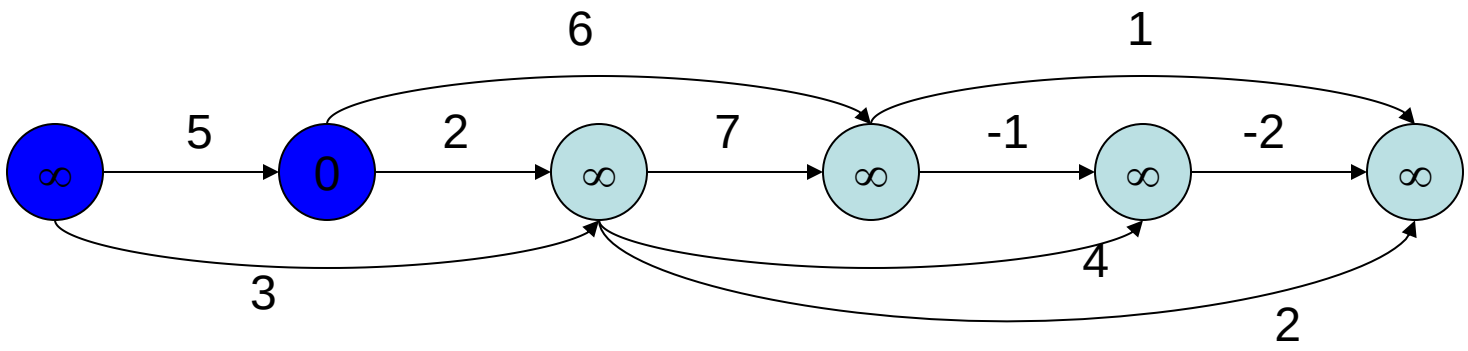
Shortest Paths in DAGs - Example



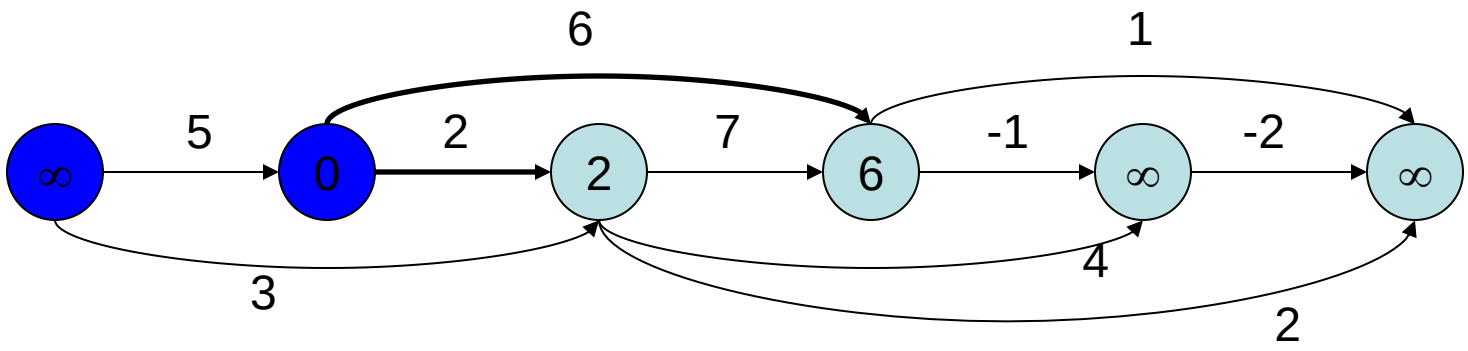
Shortest Paths in DAGs - Example



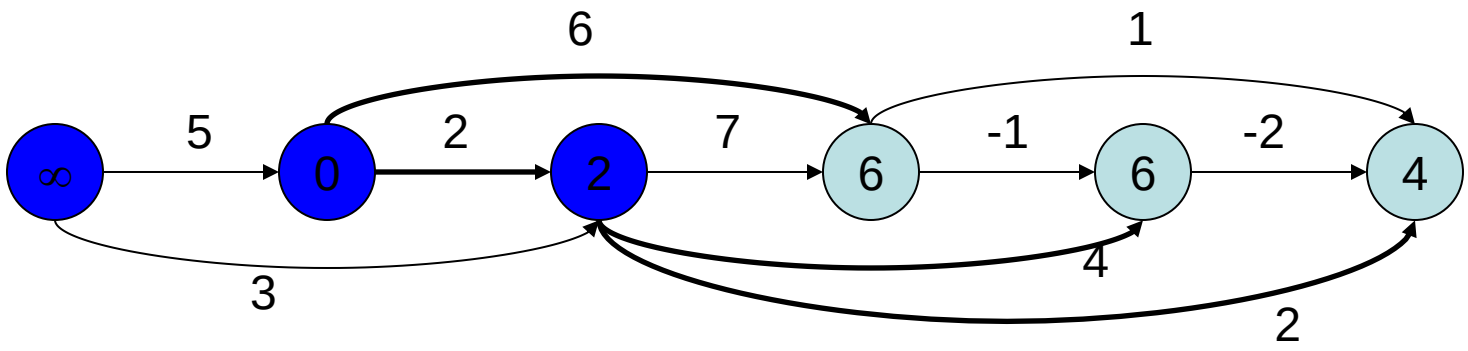
Shortest Paths in DAGs - Example



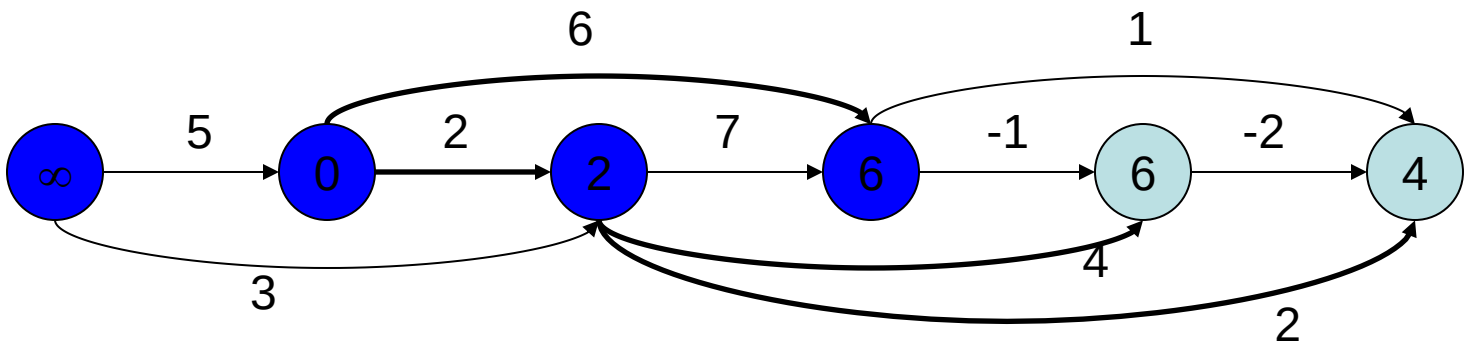
Shortest Paths in DAGs - Example



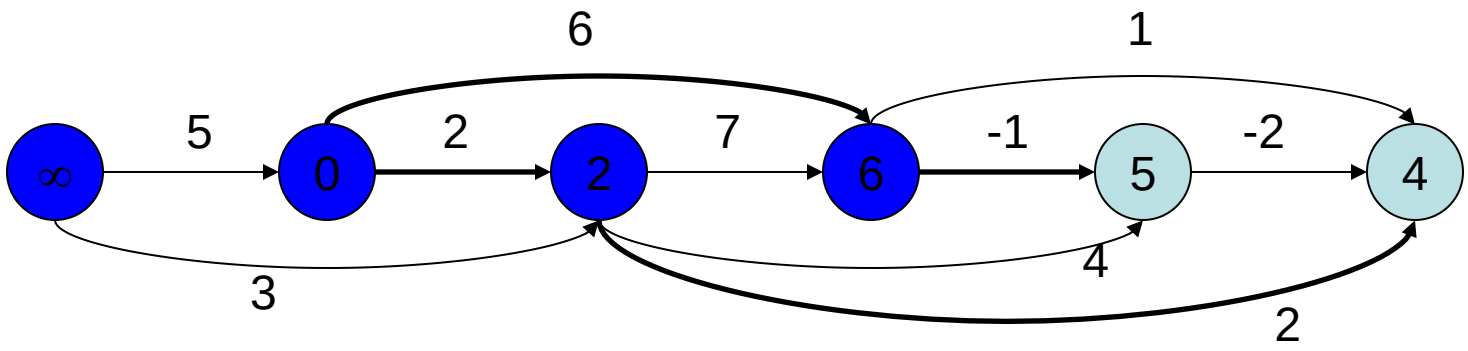
Shortest Paths in DAGs - Example



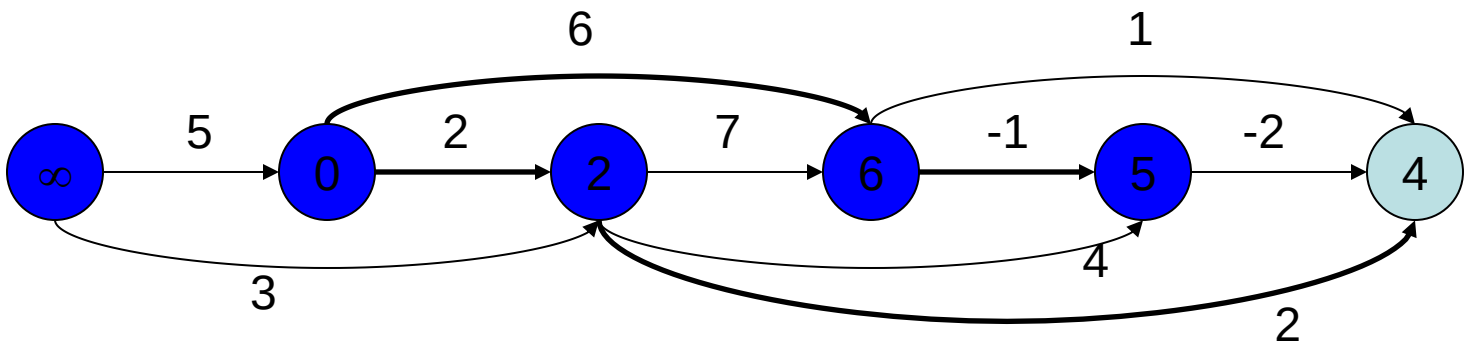
Shortest Paths in DAGs - Example



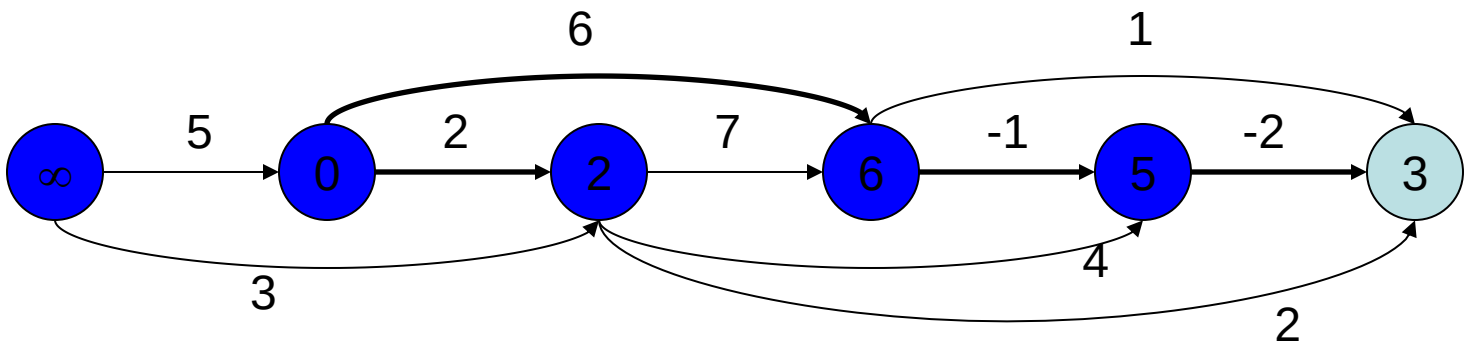
Shortest Paths in DAGs - Example



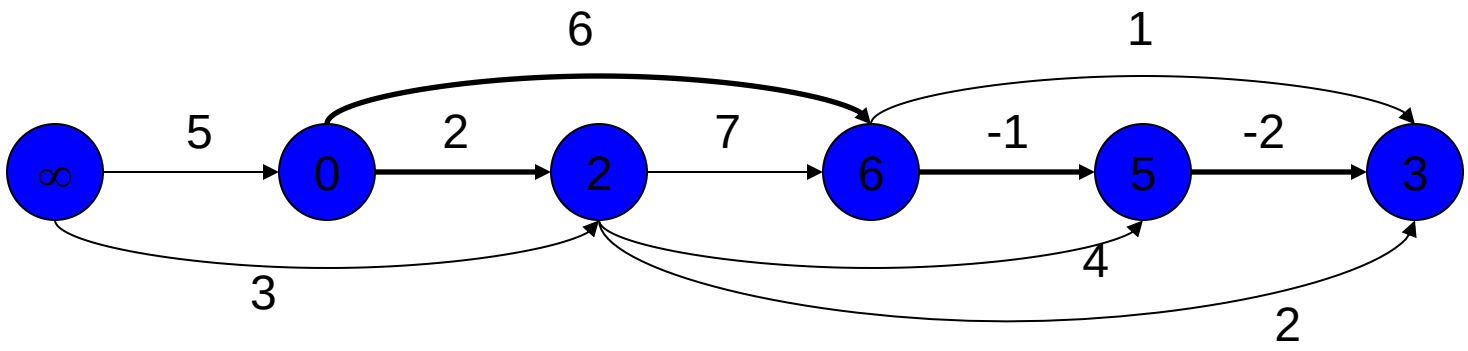
Shortest Paths in DAGs - Example



Shortest Paths in DAGs - Example



Shortest Paths in DAGs - Example



SP in a DAGs - Complexity

SP-DAG(**graph** (G,w), s)

topologically sort vertices of G

InitializeSingleSource(G, s)

for each vertex u taken in topologically sorted order **do**
for each vertex $v \in$ *Adj*[u] **do**
Relax(u,v,w)

$T(V,E) = \Theta(V + E) + \Theta(V) + \Theta(V) + E\ \Theta(1) = \Theta(V + E)$