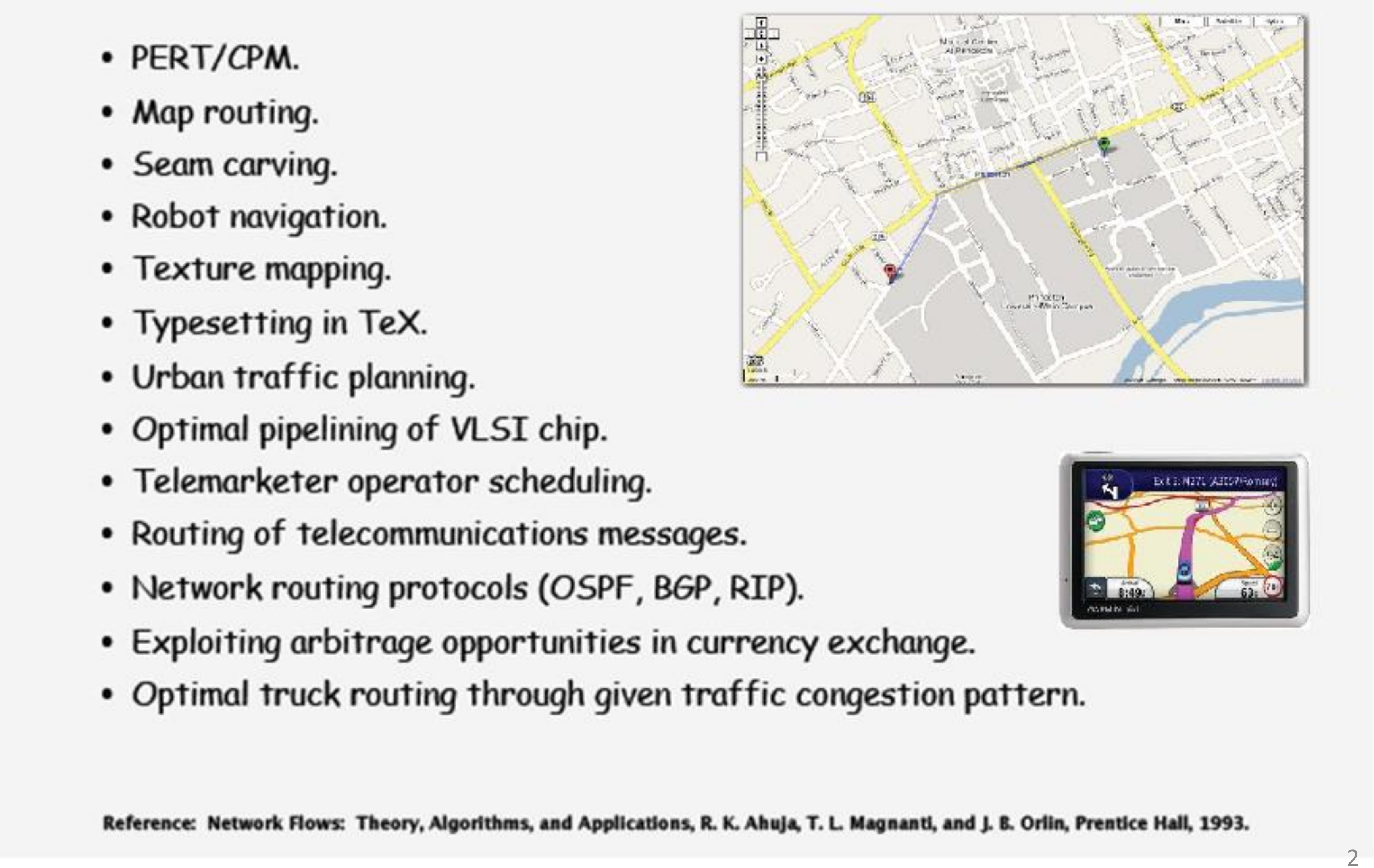
COMP 355
Advanced Algorithms

Bellman-Ford Algorithm for Shortest Paths
Sections 6.8 (KT)

Rhodes College
—1848—

Shortest Path Applications



- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

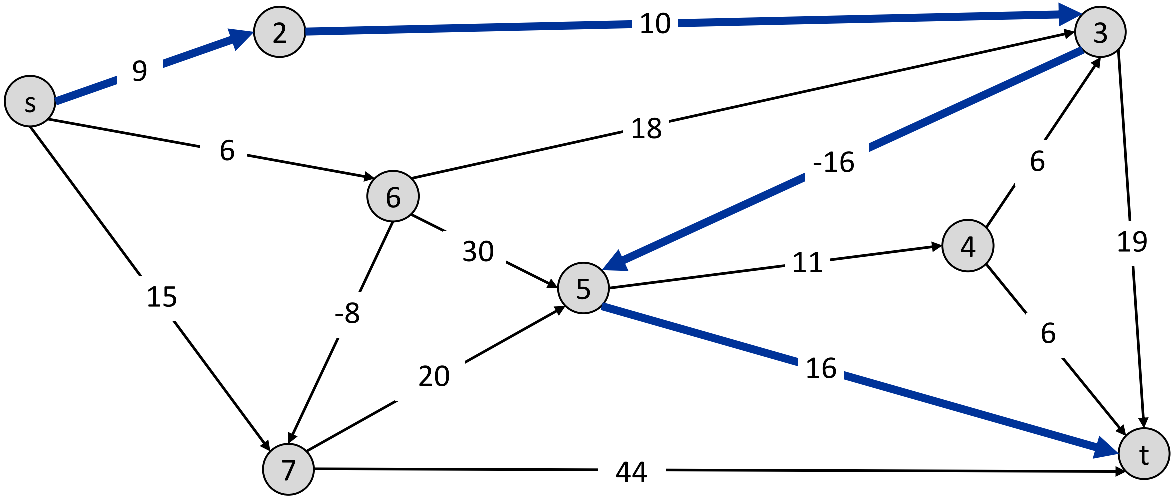
Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest Paths

Shortest path problem. Given a directed graph G = (V, E), with edge weights ℓ_{vw}, find shortest path from node s to node t.

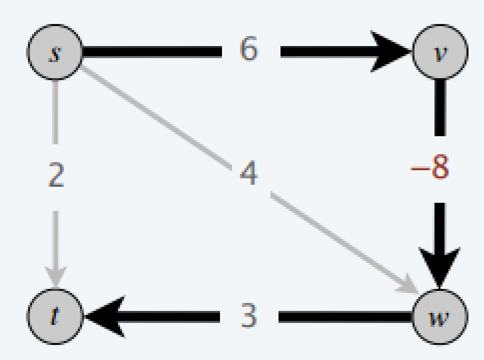
allow negative weights

Ex. Nodes represent agents in a financial setting and ℓ_{vw} is cost of transaction in which we buy from agent v and sell immediately to w.



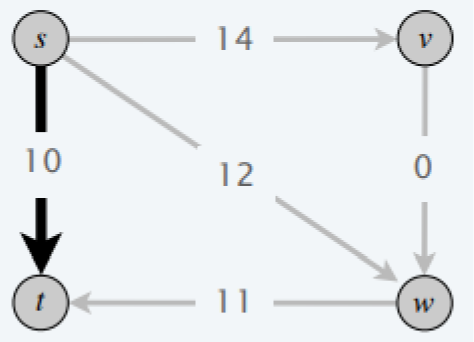
Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs.



Dijkstra selects the vertices in the order s, t, w, v
But shortest path from s to t is $s \to v \to w \to t$.

Re-weighting. Adding a constant to every edge weight can fail.

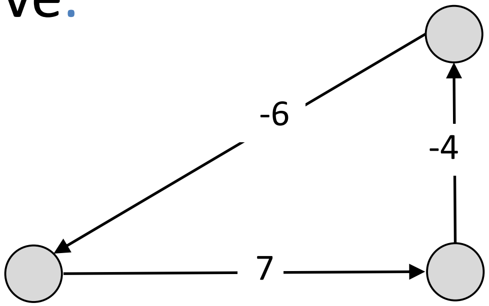


Adding 8 to each edge weight changes the shortest path from $s \to v \to w \to t$ to $s \to t$.

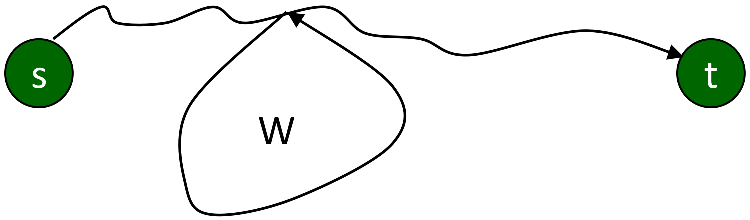
Bad news: Need a different algorithm

Shortest Paths: Negative Cost Cycles

Def. A negative cycle is a directed cycle such that the sum of its edge weights is negative.



Observation. If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple.



$\ell(W) < 0$

Dynamic Programming Subproblems

Which subproblems to find shortest v↝t paths for every node v?

A. $OPT(i, v)$ = length of shortest v↝t path that uses exactly i edges.

B. $OPT(i, v)$ = length of shortest v↝t path that uses at most i edges.

C. Neither A nor B

Shortest Paths: Dynamic Programming

Def. $OPT(i, v)$ = length of shortest $v - t$ path P using at most i edges.

Goal: $OPT(n-1, v)$ for each v.

if no negative cycles, there exists a shortest v$\leadsto$t path that is simple

- Case 1: P uses at most $i - 1$ edges.
  - $OPT(i, v) \; = \; OPT(i-1, v)$
- Case 2: P uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w) (cost of ℓ_{vw})
  - Then selects best $w - t$ path using at most $i - 1$ edges

$$OPT(i,v) \;=\; \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\left\{ OPT(i-1,v), \; \min\limits_{(v,w)\in E} \{OPT(i-1,w) + \ell_{vw}\} \right\} & \text{if } i > 0 \end{cases}$$

Shortest Paths: Implementation

```
Shortest-Path(V, E, $\ell$, t) {
   foreach node v $\in$ V
      M[0, v] $\leftarrow$ $\infty$
   M[0, t] $\leftarrow$ 0

   for i = 1 to n-1
      foreach node v $\in$ V
         M[i, v] $\leftarrow$ M[i-1, v]
      foreach edge (v, w) $\in$ E
         M[i, v] $\leftarrow$ min { M[i, v], M[i-1, w] + $\ell_{vw}$ }
}
```

- Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space.

- Finding the shortest paths. Maintain a "successor" for each table entry.

Bellman-Ford: Practical Improvements

Space Optimization: Maintain two 1D arrays (instead of 2D array).

- $d[v]$ = length of a shortest v↝t path that we have found so far.
- $successor[v]$ = next node on a v↝t path.

Performance Optimization: If d$[w]$ does not change during iteration $i-1$, no need to relax any edge pointing to w in iteration i

Bellman-Ford: Efficient Implementation

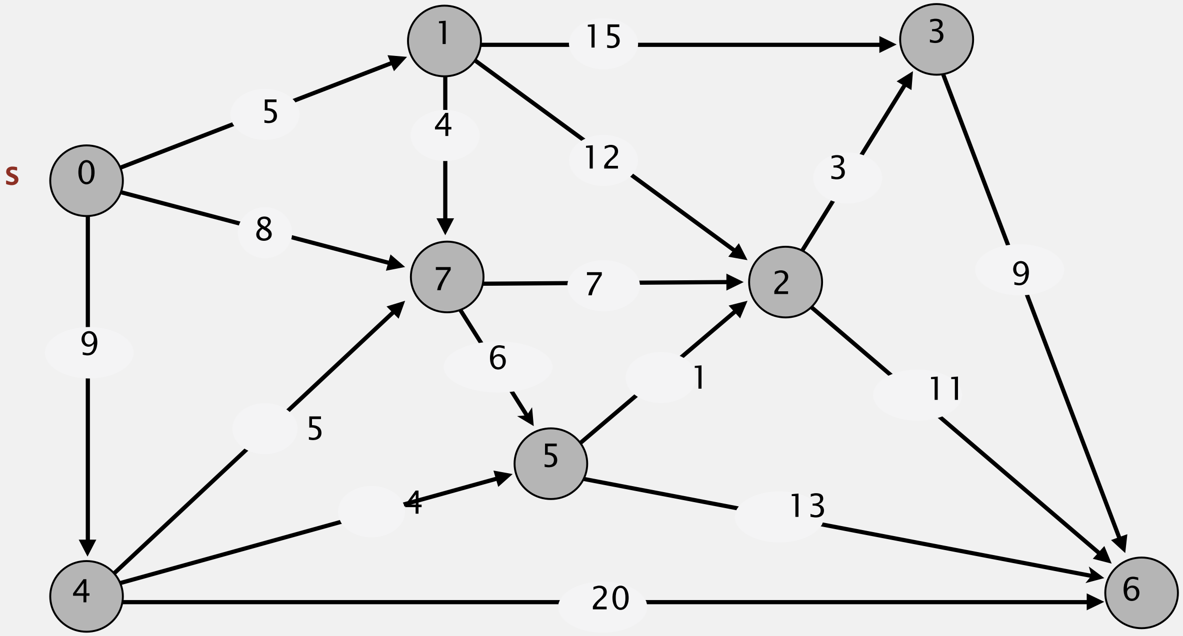
```
Shortest-Path(V, E, $\ell$, t) {
   foreach node v ∈ V
      d[v] ← ∞
      successor[v] ← NULL
   d[t] ← 0

   for i = 1 to n-1
      foreach node w ∈ V
         if (d[w] was updated in previous pass)
            foreach edge (v, w) ∈ E
               if (d[v] > d[w] + $\ell_{vw}$)
                  d[v] ← d[w] + $\ell_{vw}$
                  successor[v] ← w
      if (no d[*] value changed in pass i) STOP
}
```

pass i
$O(m)$ time

Bellman-Ford algorithm

Repeat V times: relax all E edges.

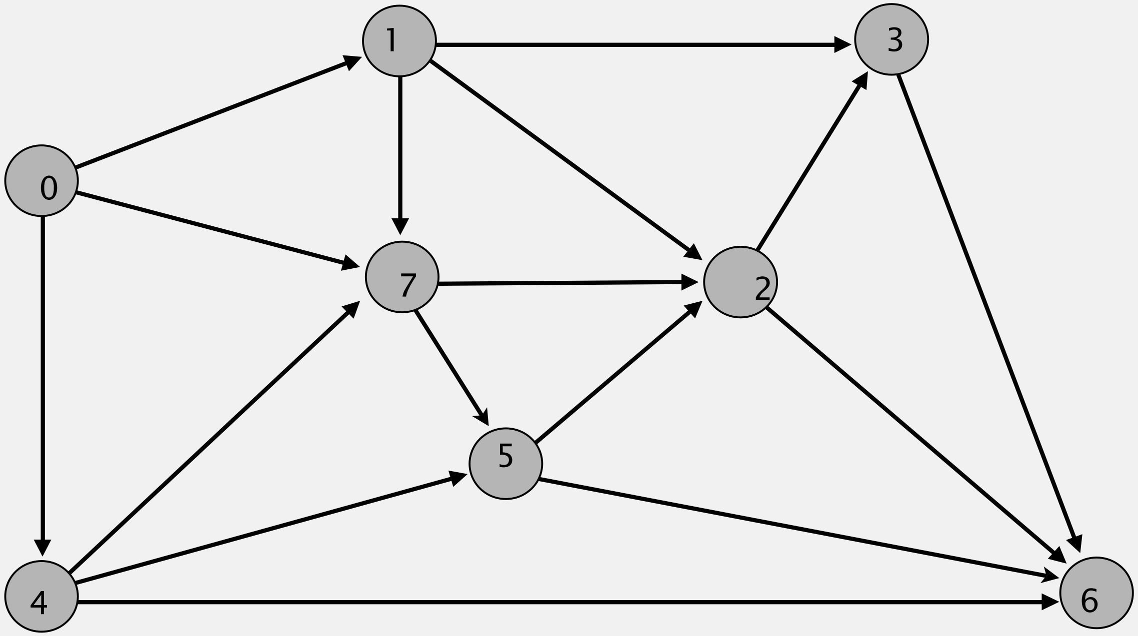


an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Bellman-Ford algorithm

Repeat V times: relax all E edges.

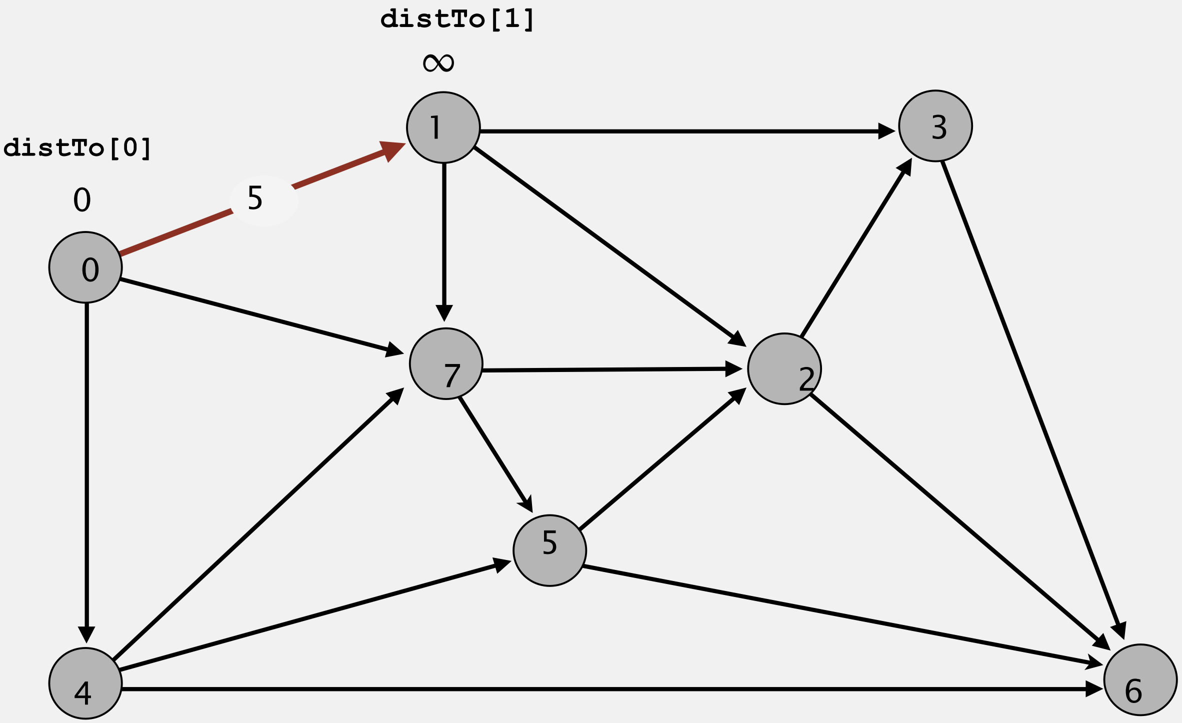


v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

initialize

Bellman-Ford algorithm

Repeat V times: relax all E edges.



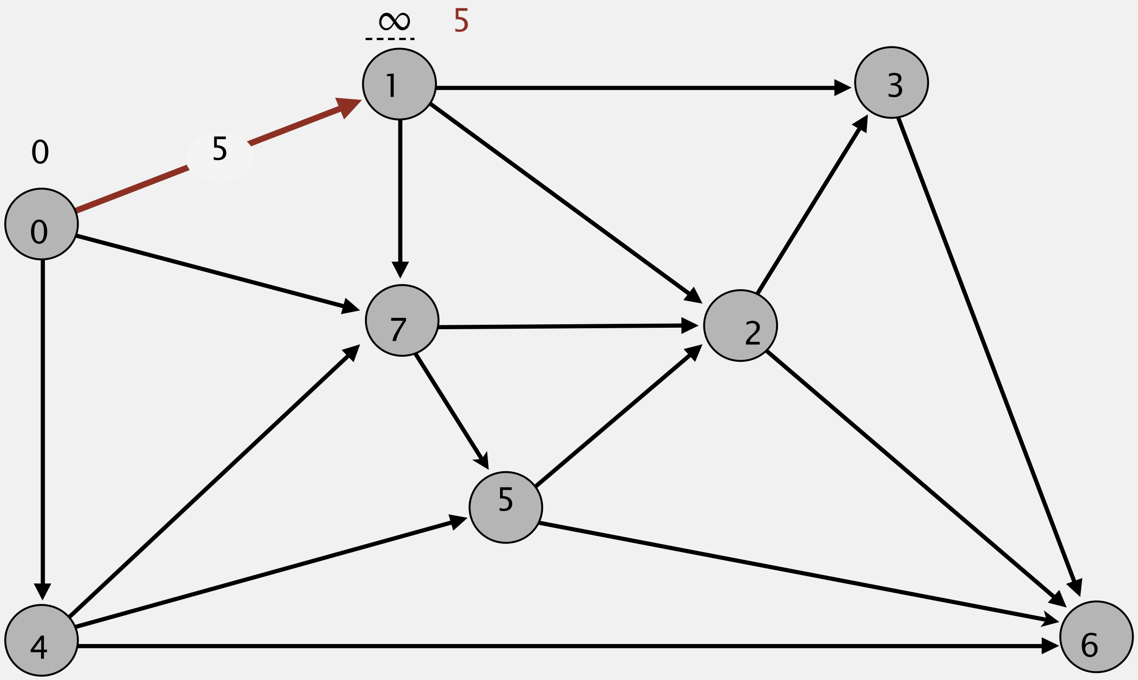
v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



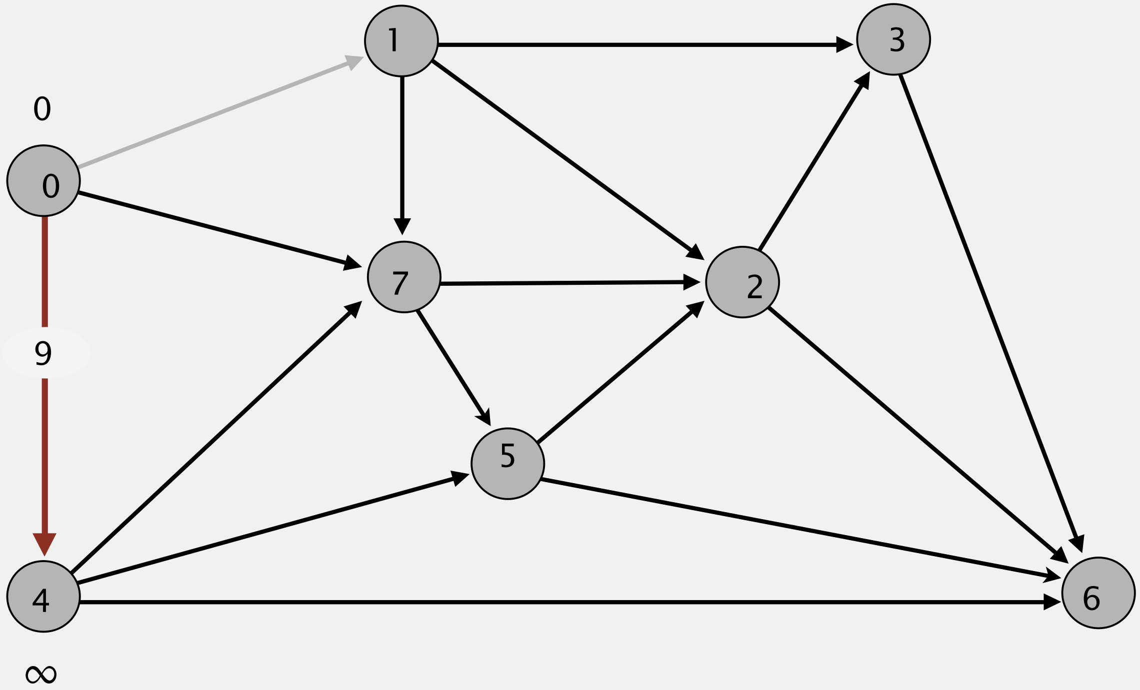
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

Bellman-Ford algorithm

Repeat V times: relax all E edges.



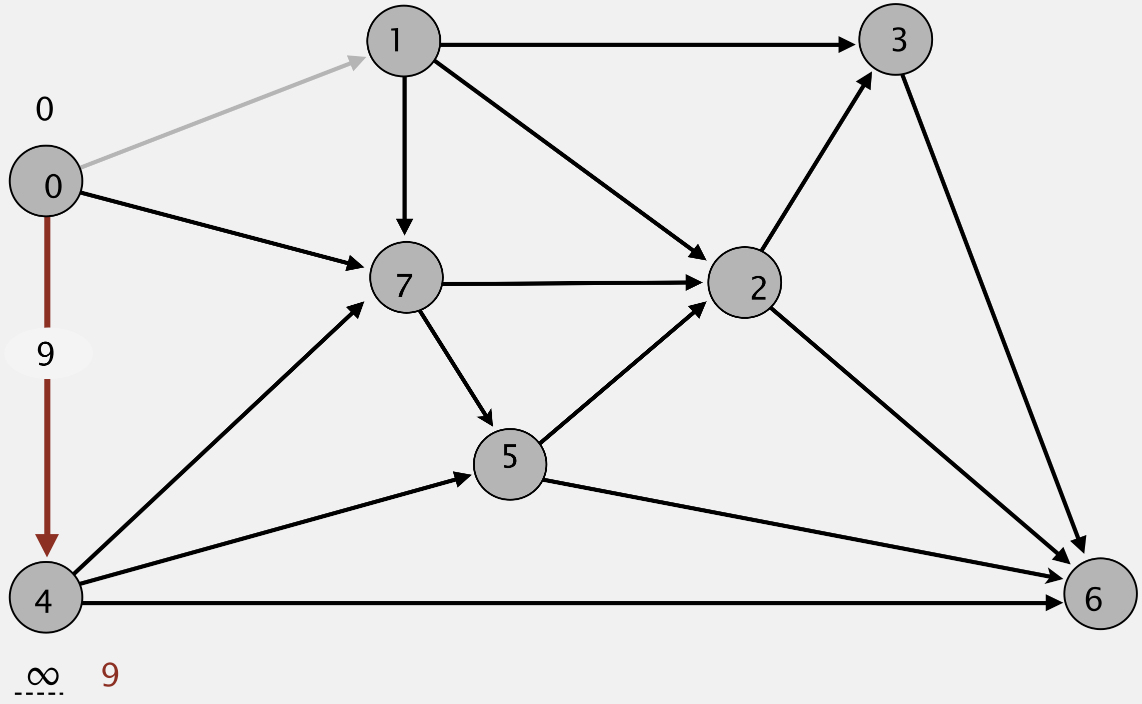
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

Bellman-Ford algorithm

Repeat V times: relax all E edges.



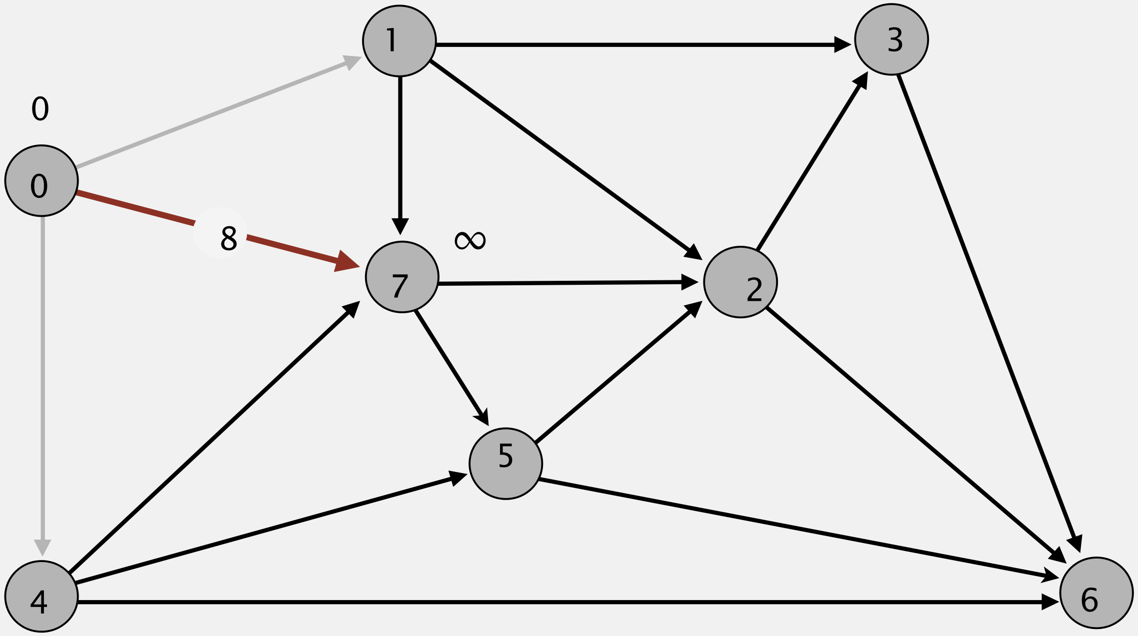
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



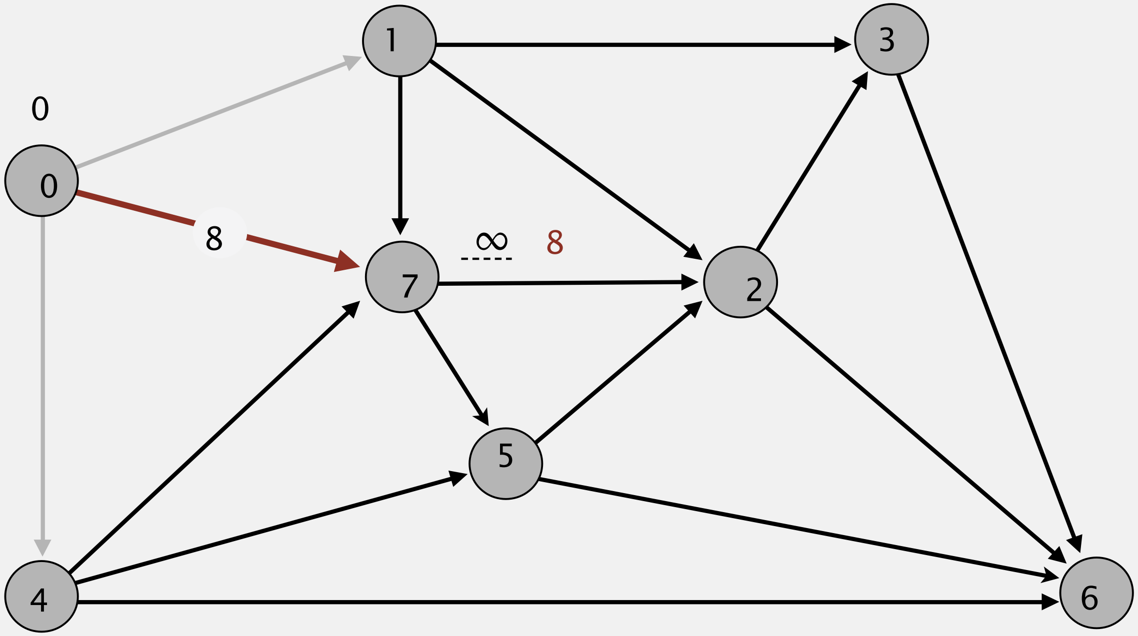
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

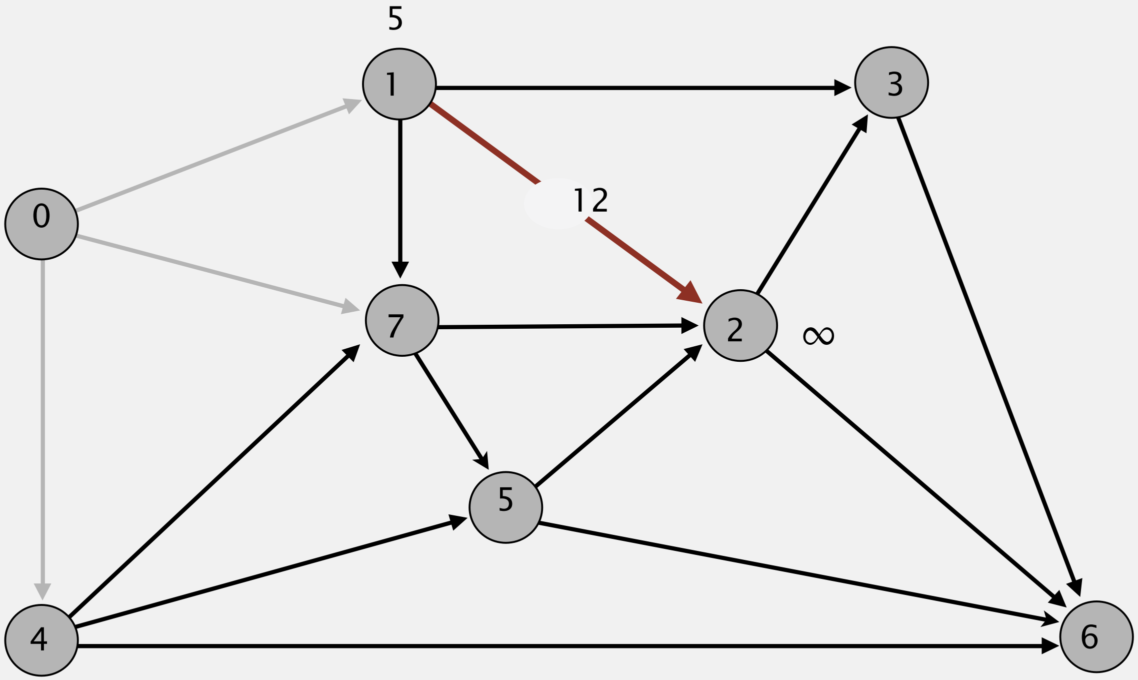
pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



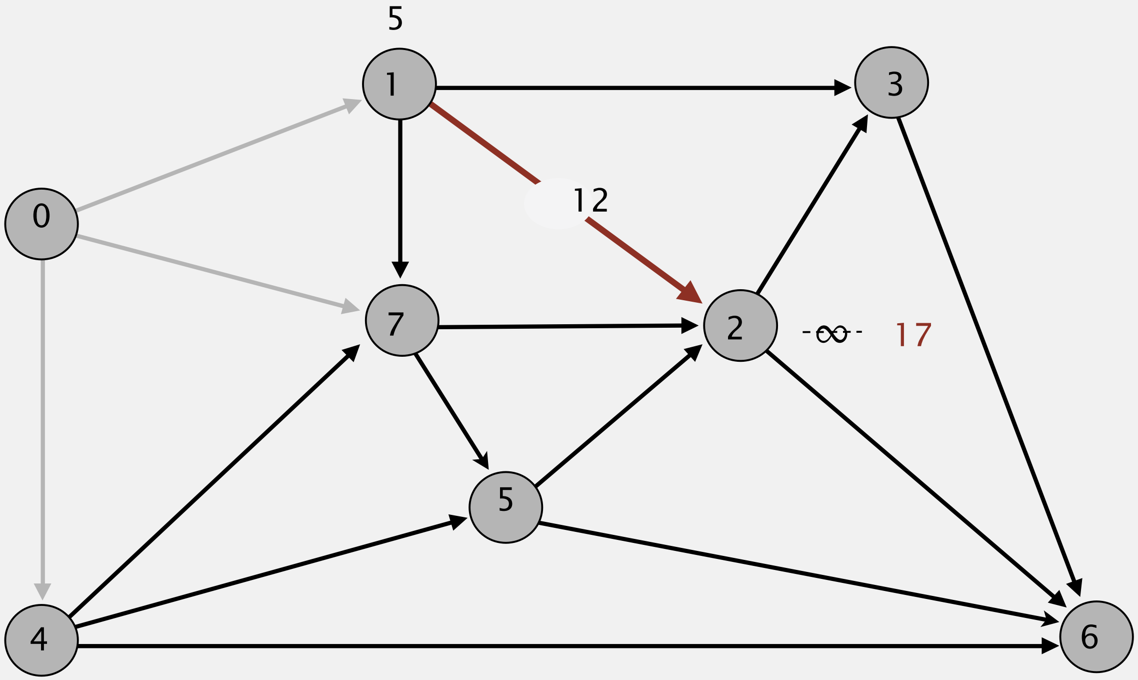
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



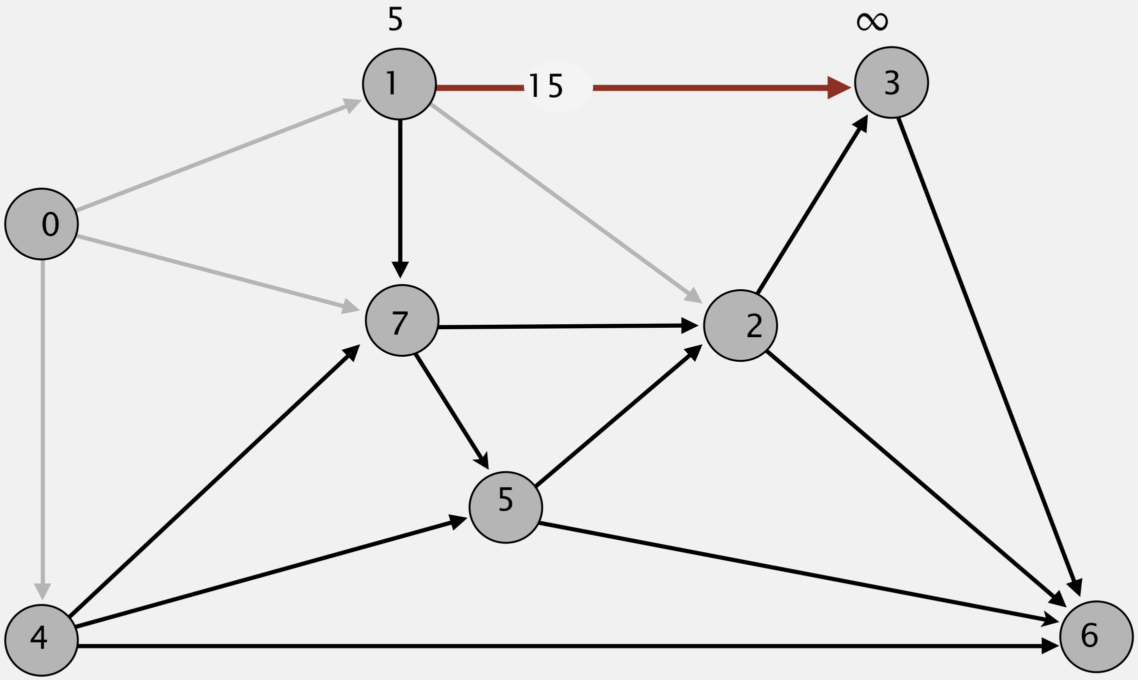
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

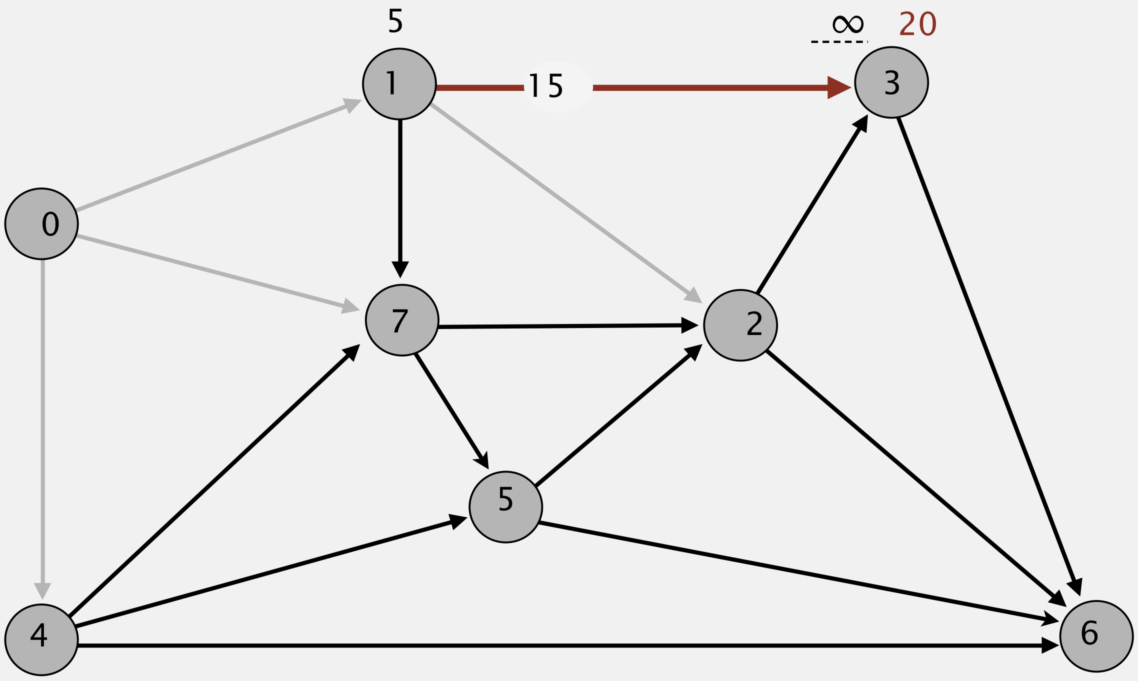
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

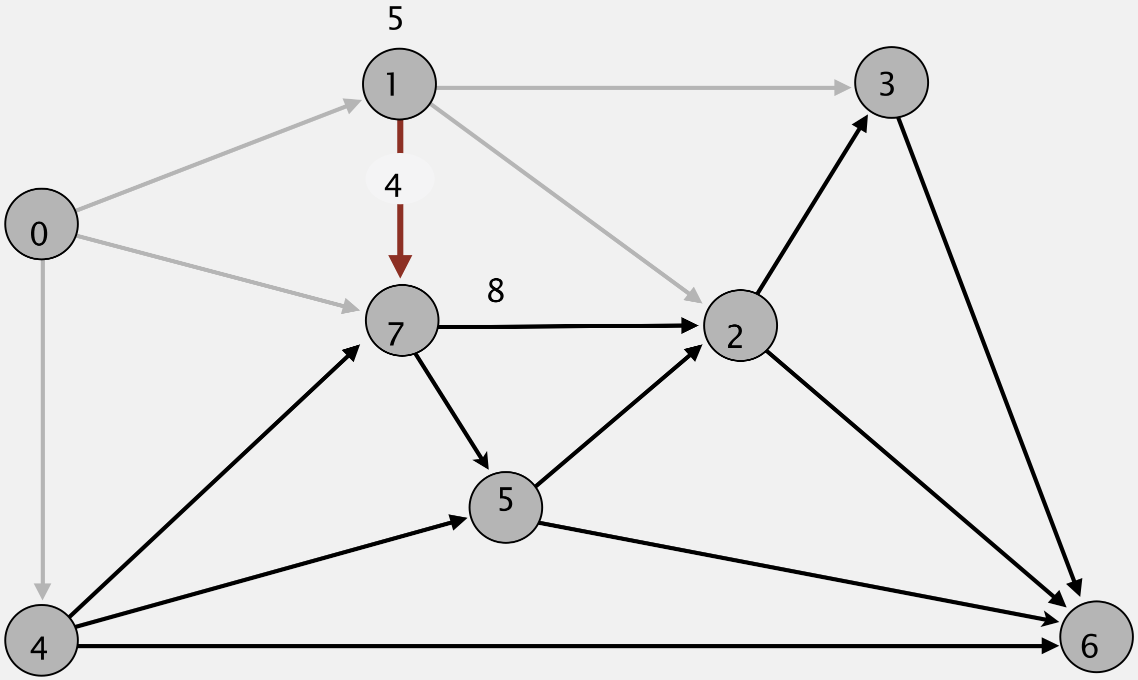
pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

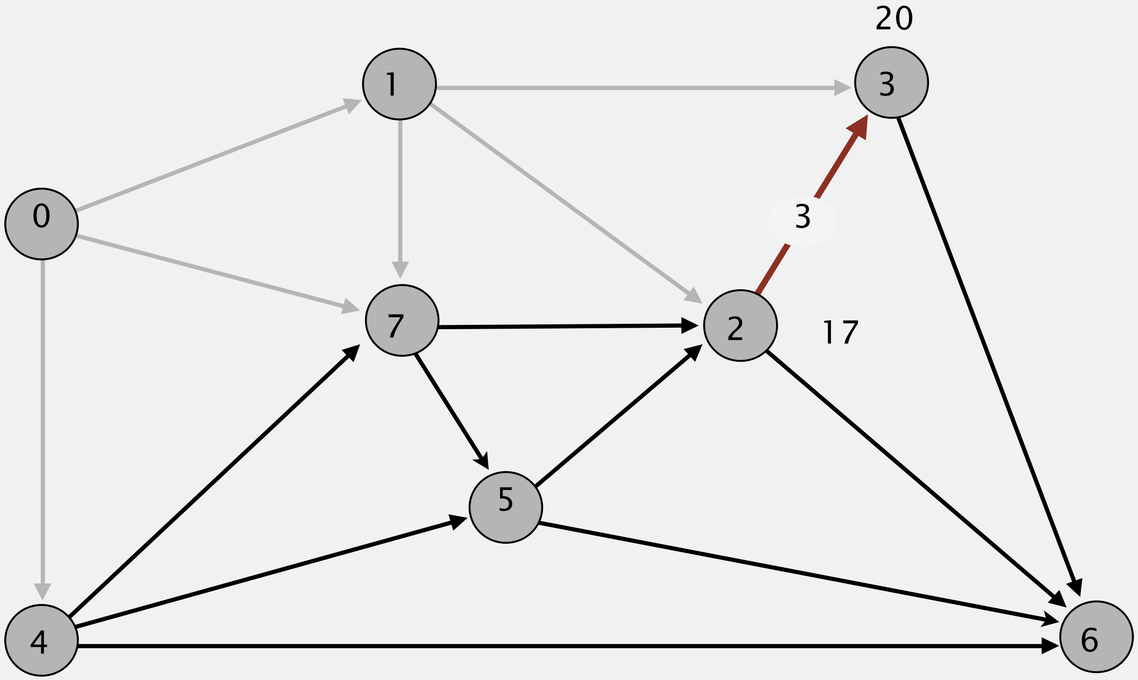
pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

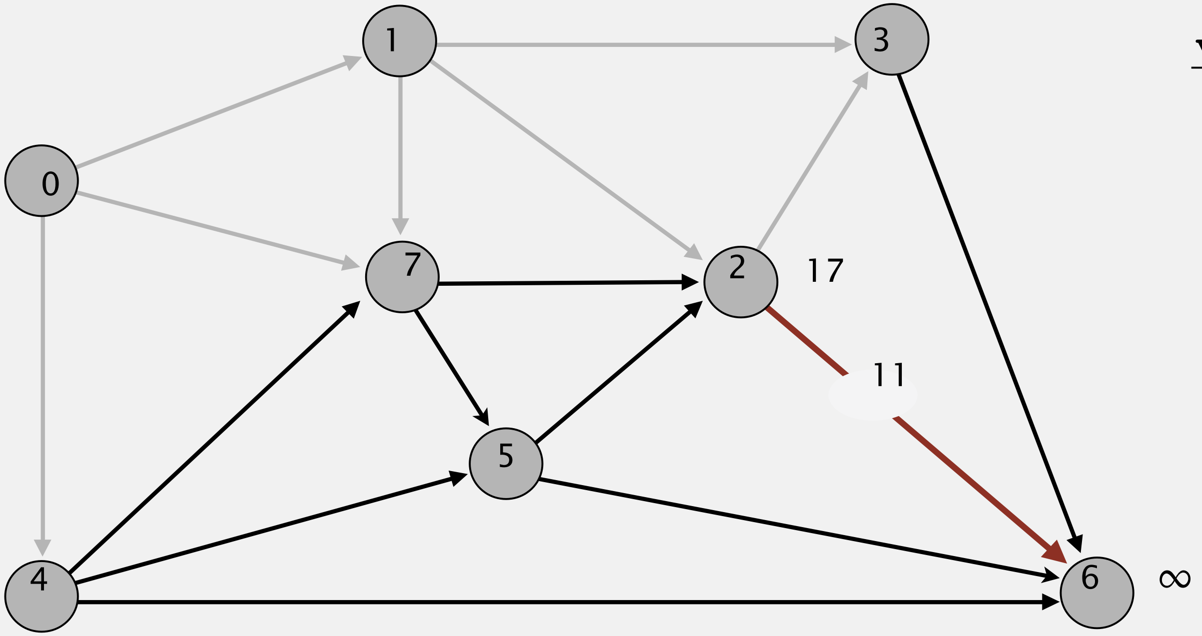
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

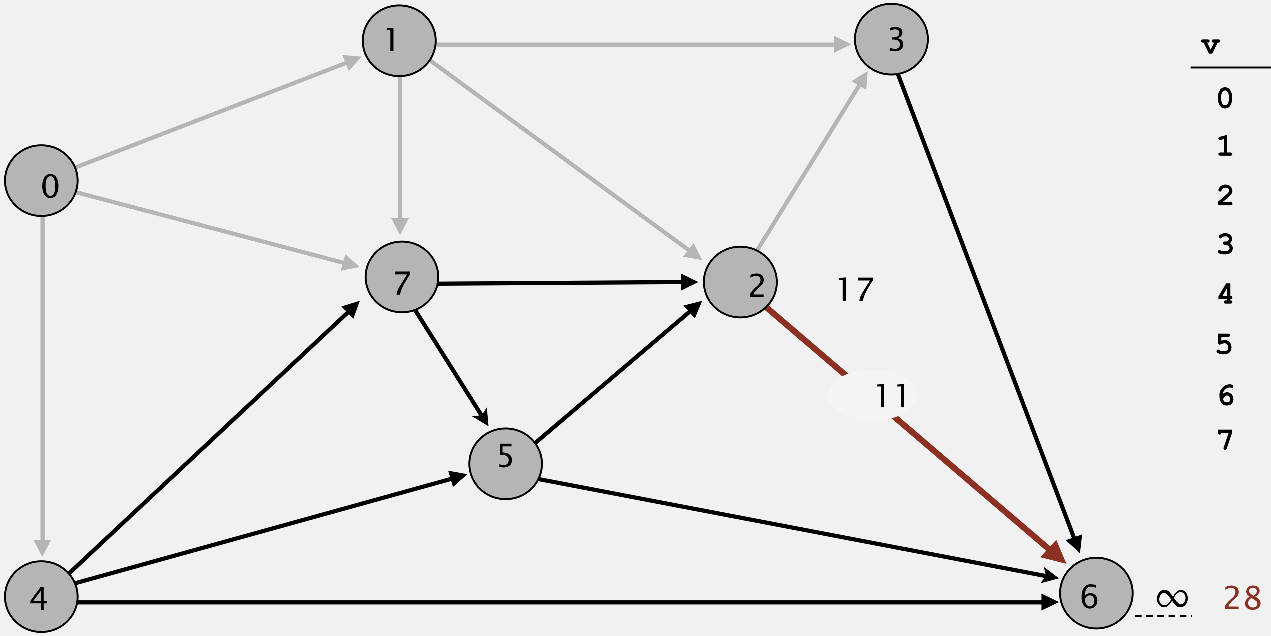
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

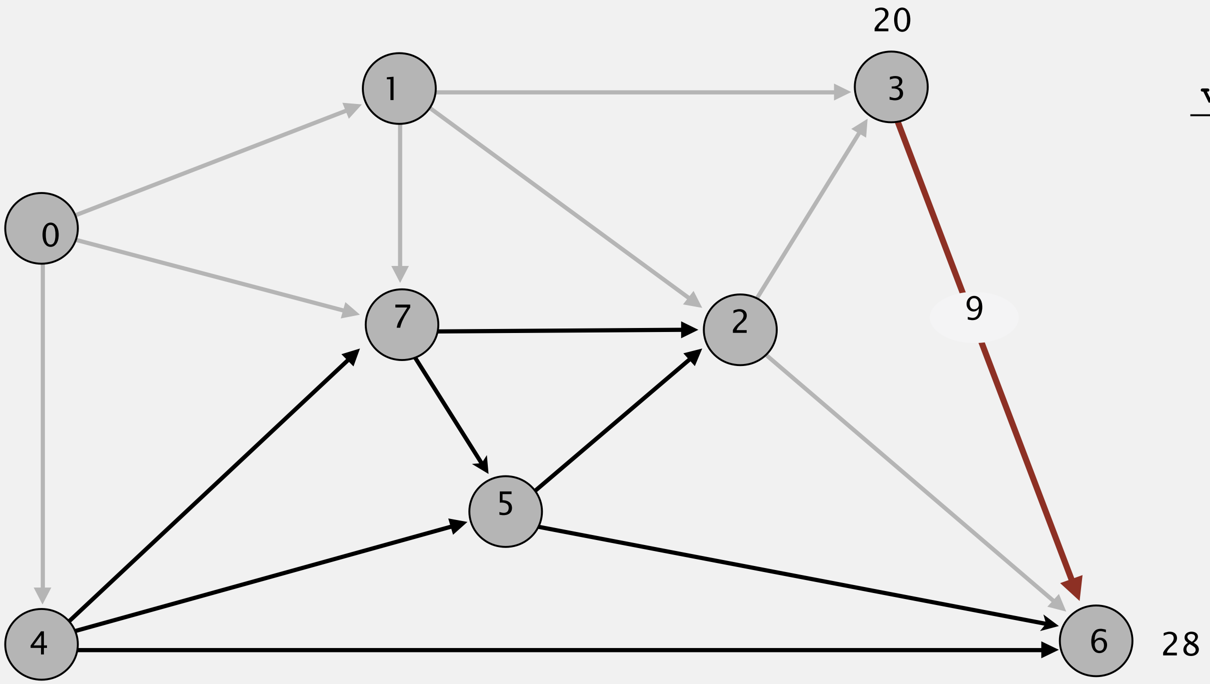
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



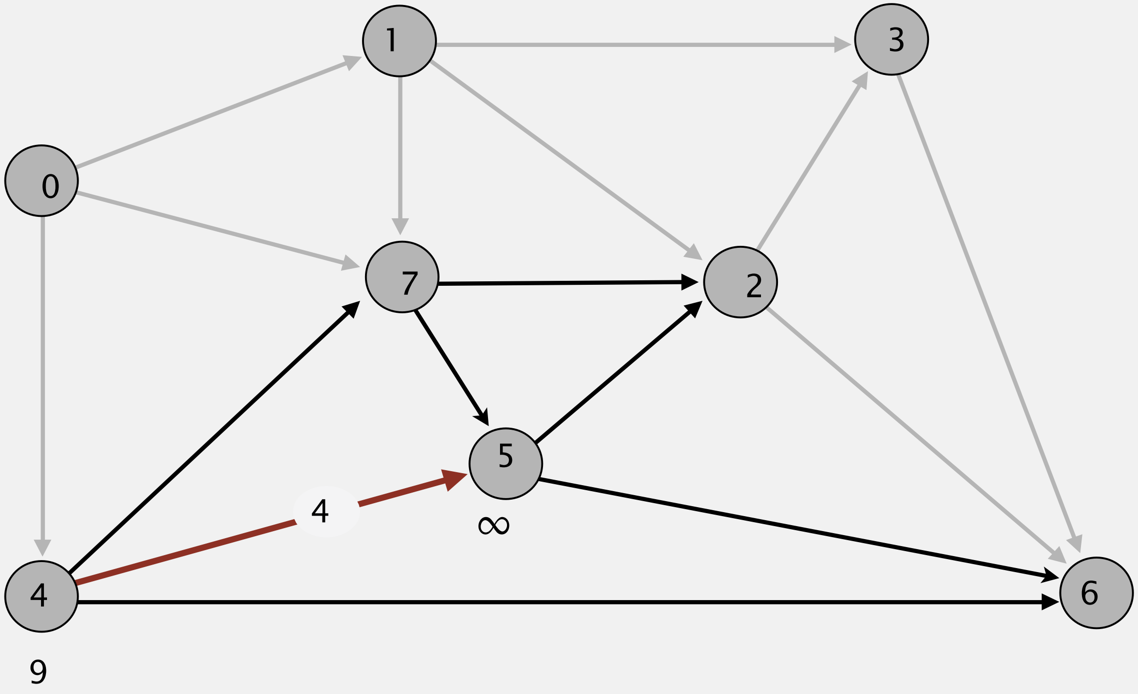
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

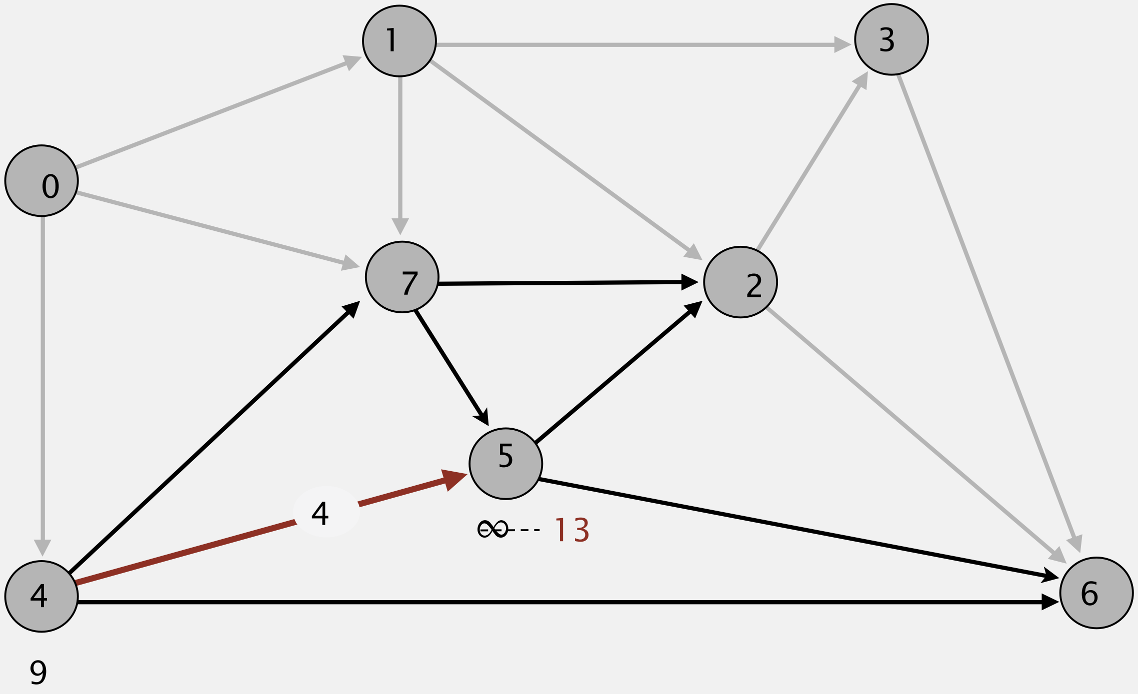
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

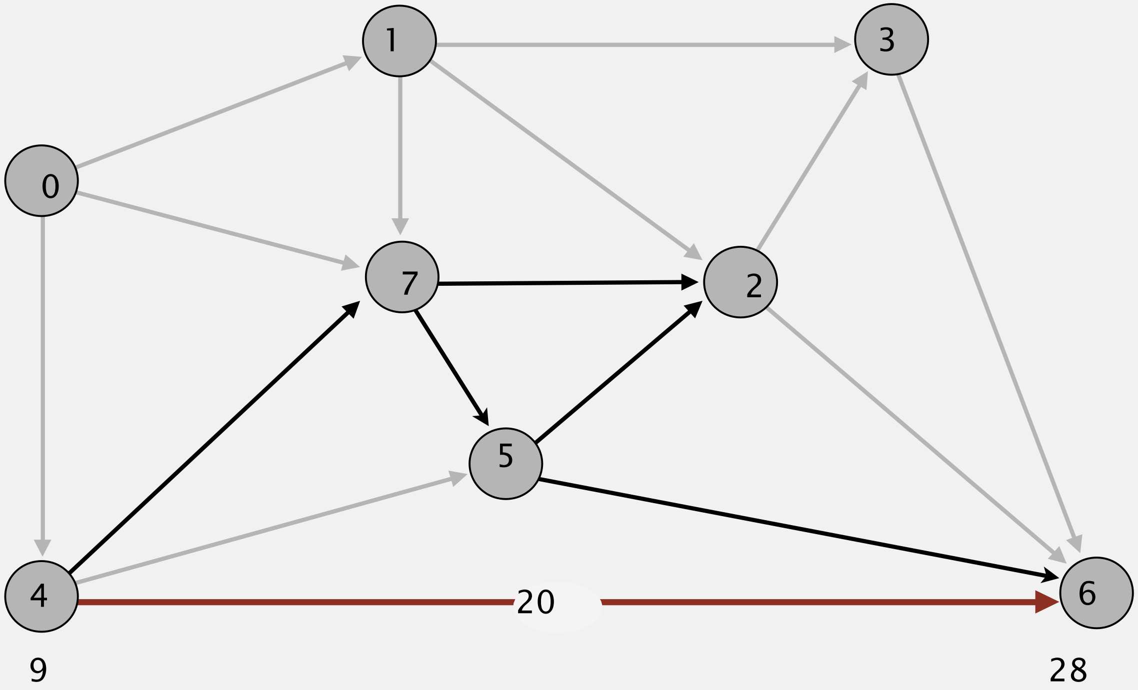
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

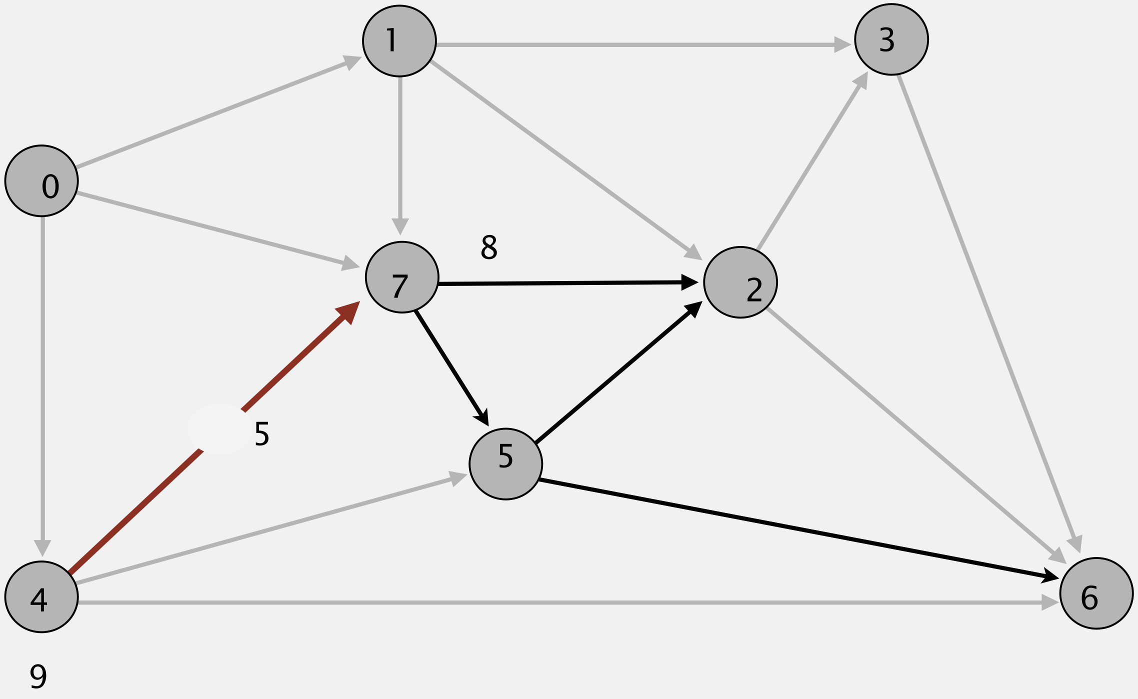
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

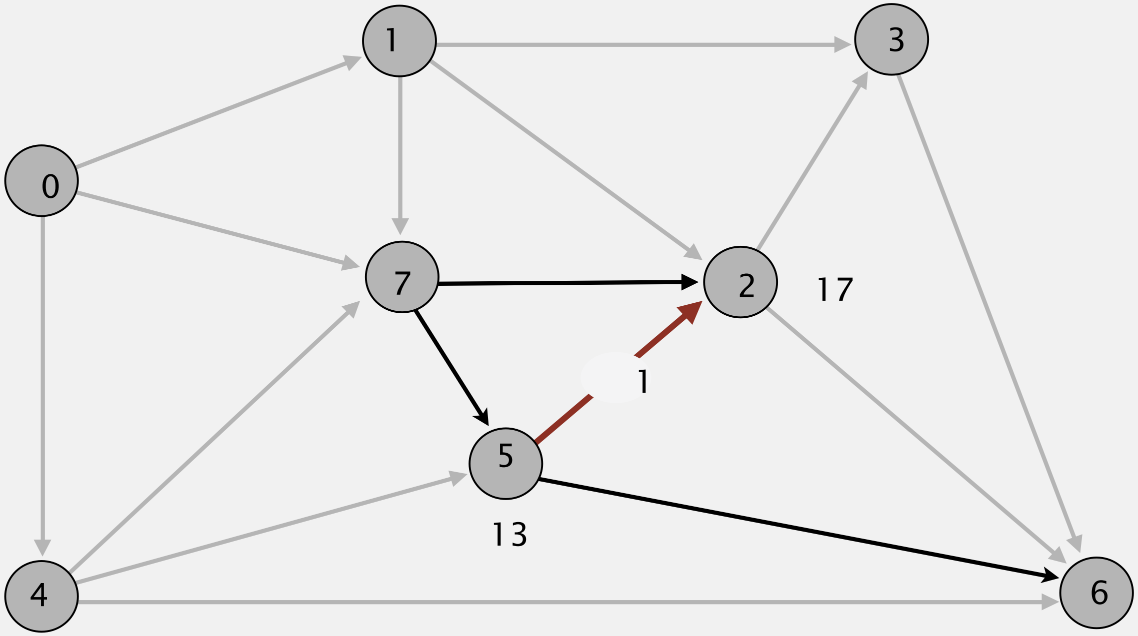
pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

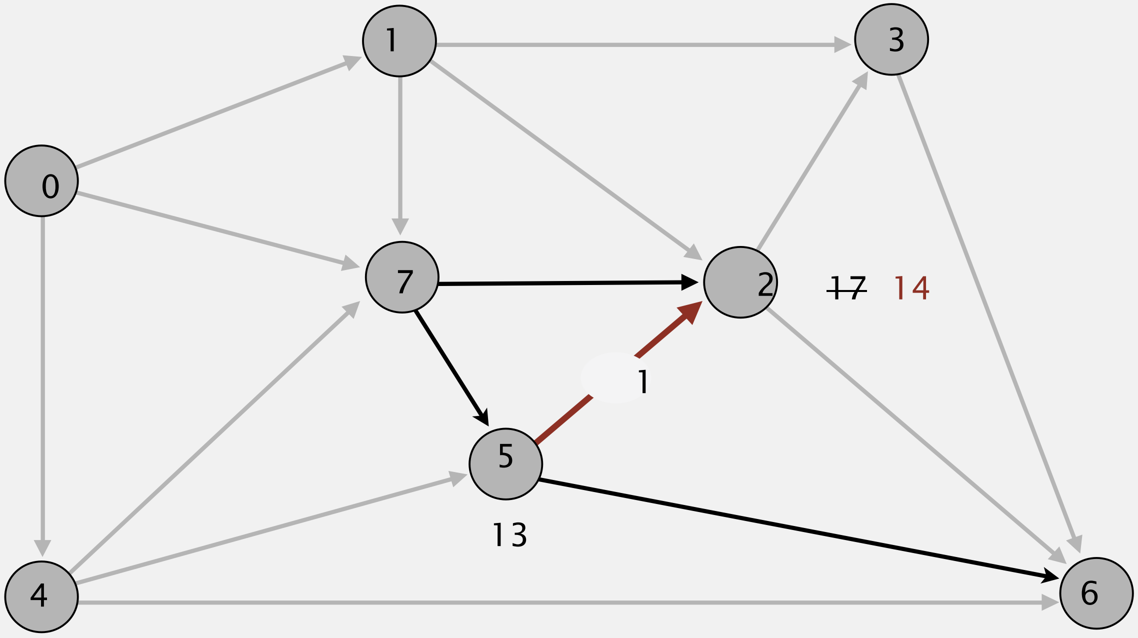
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

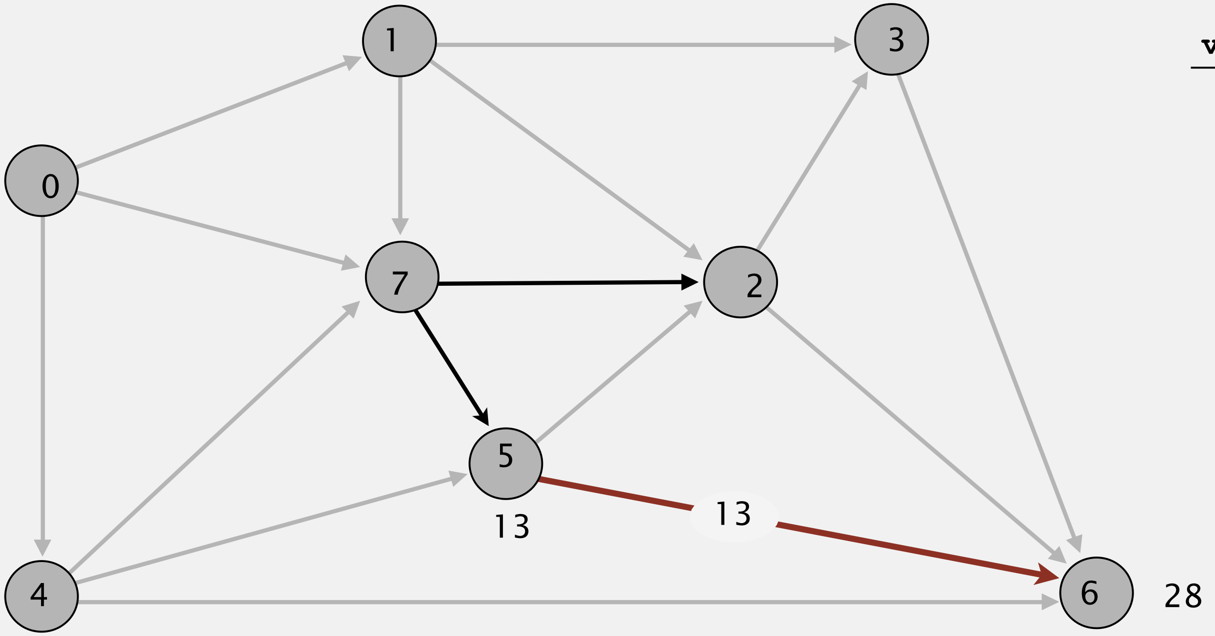
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

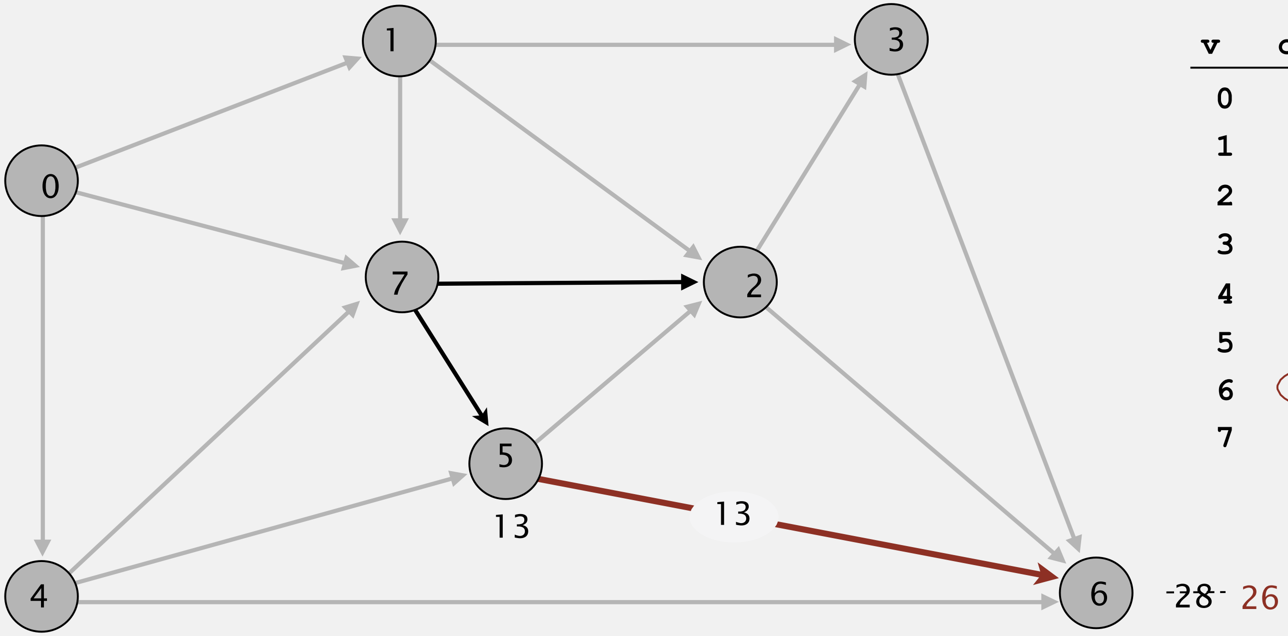
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

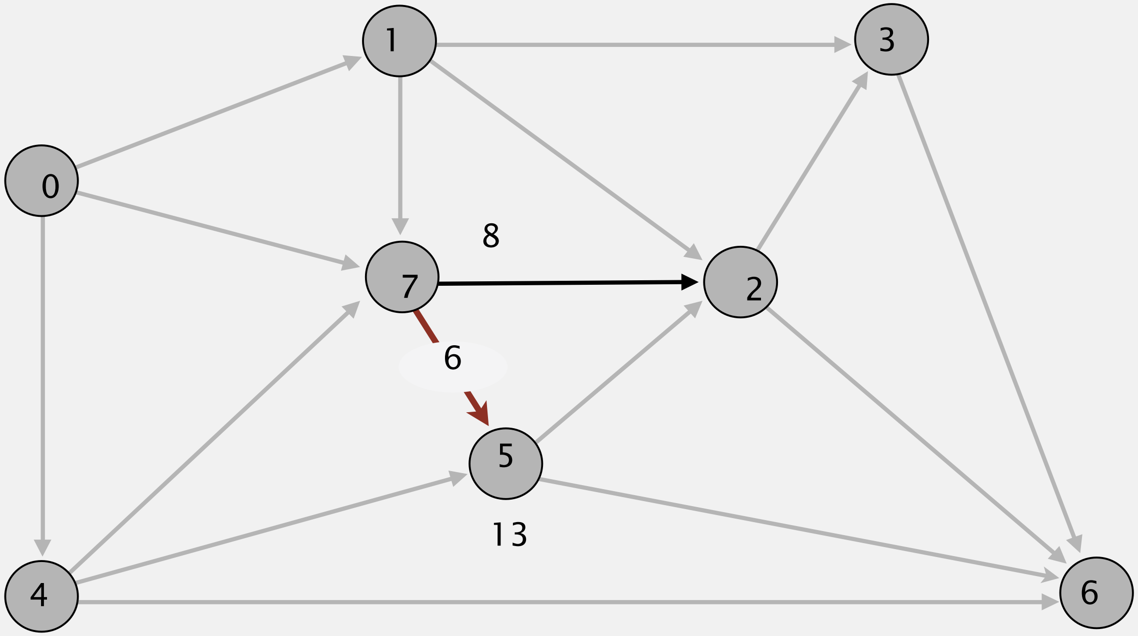
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

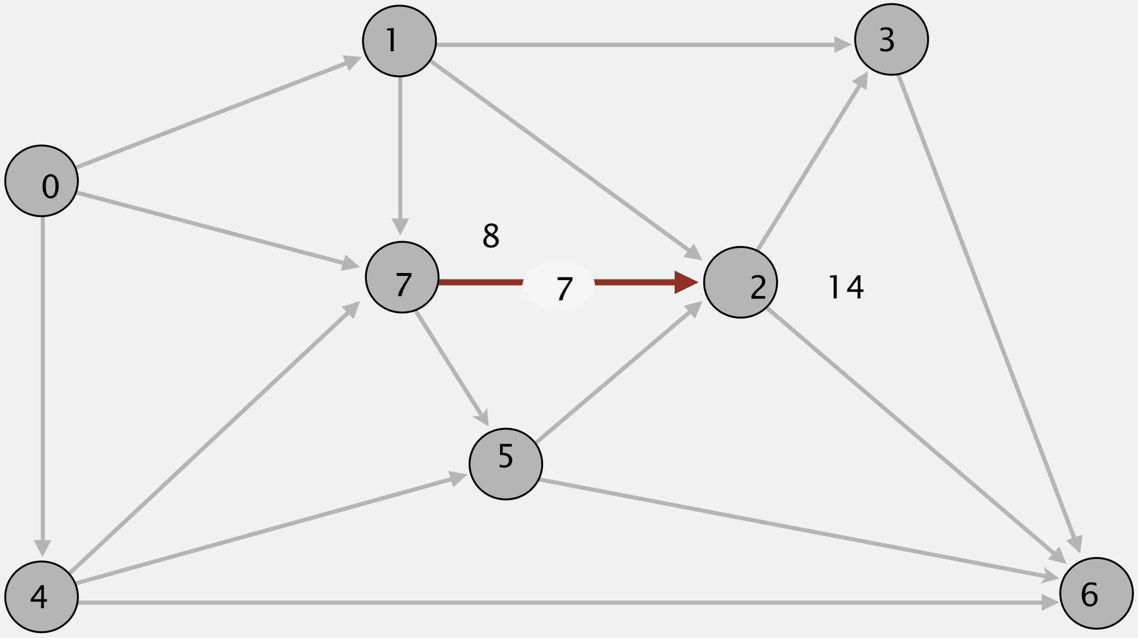
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



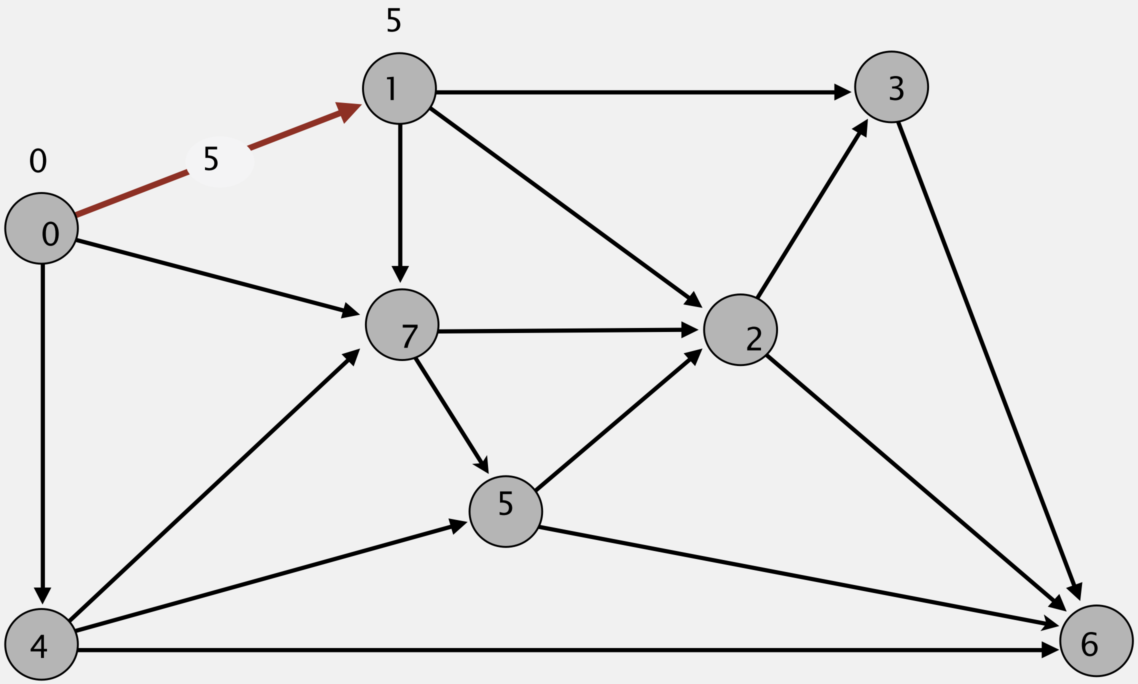
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 0

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

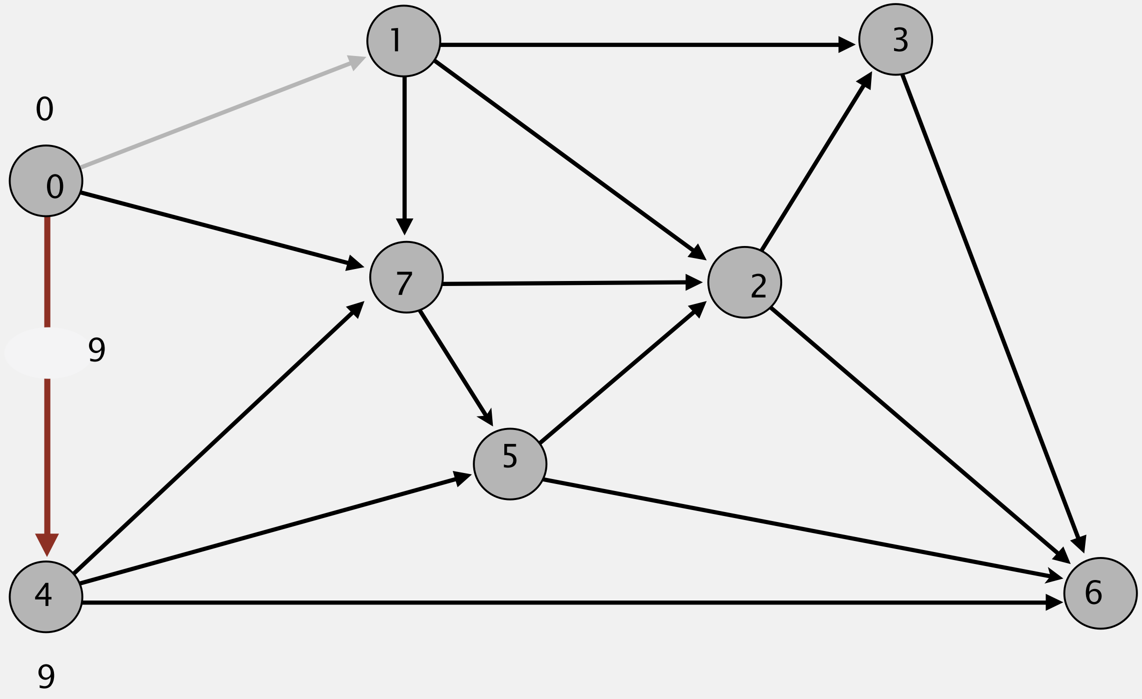
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

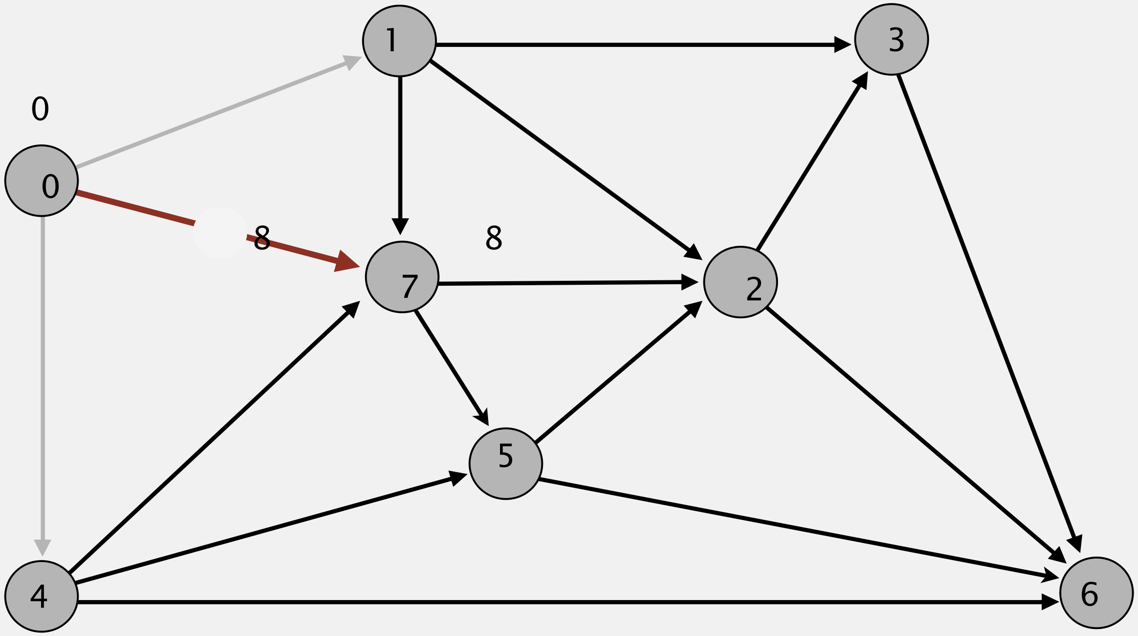
pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

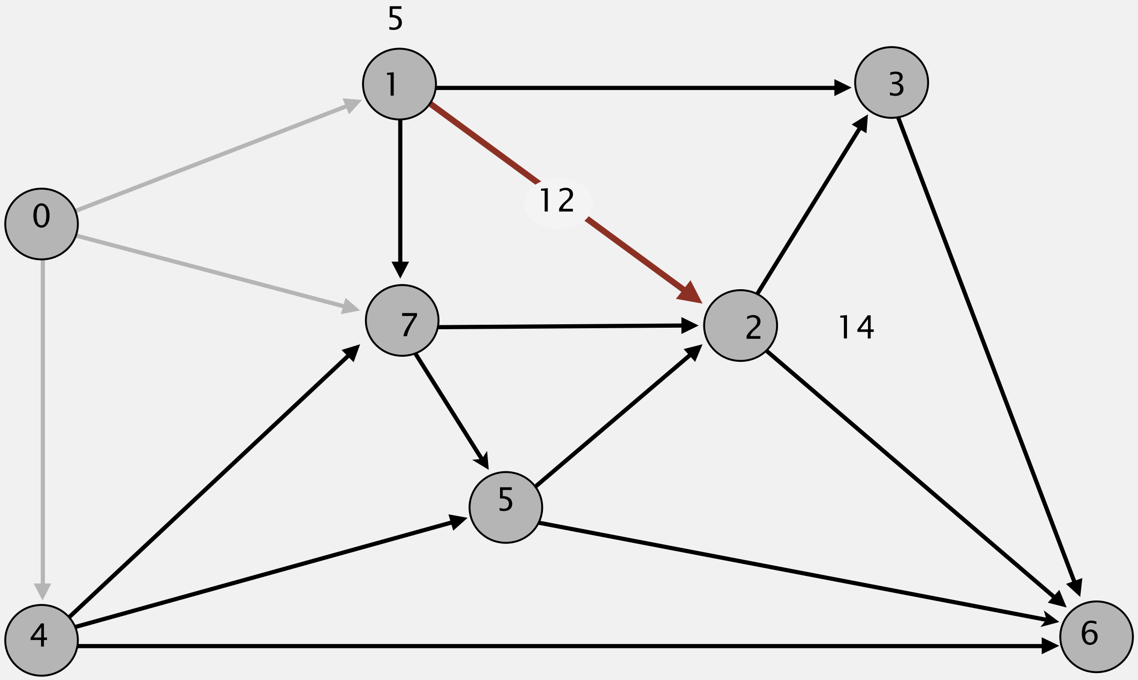
pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

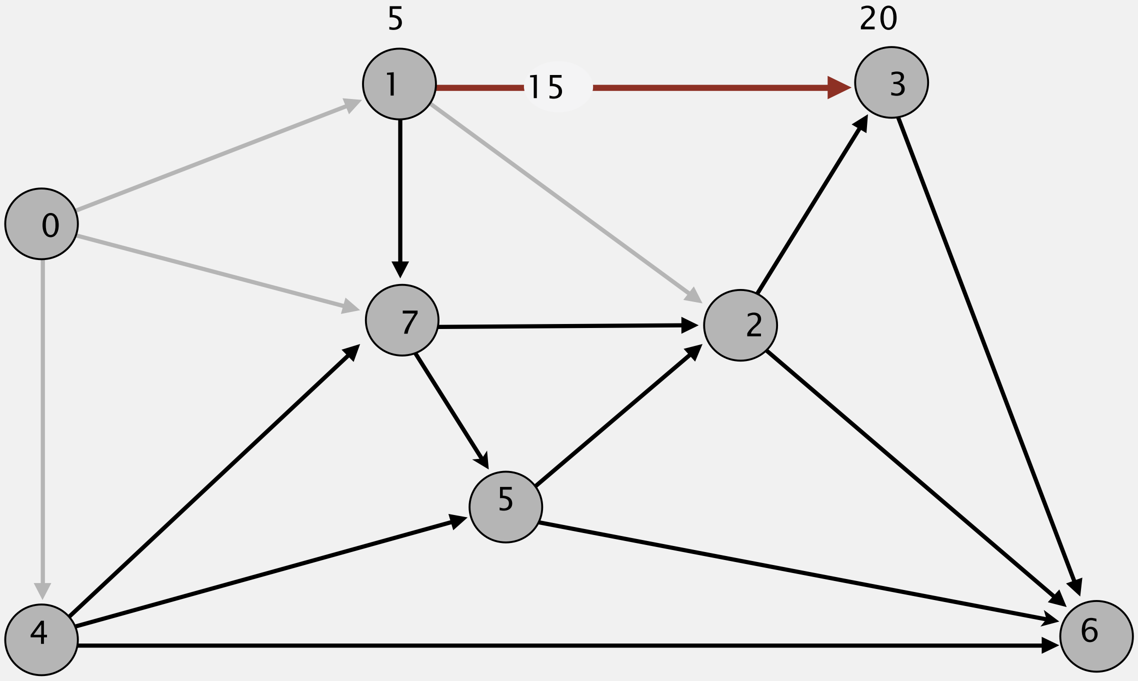
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

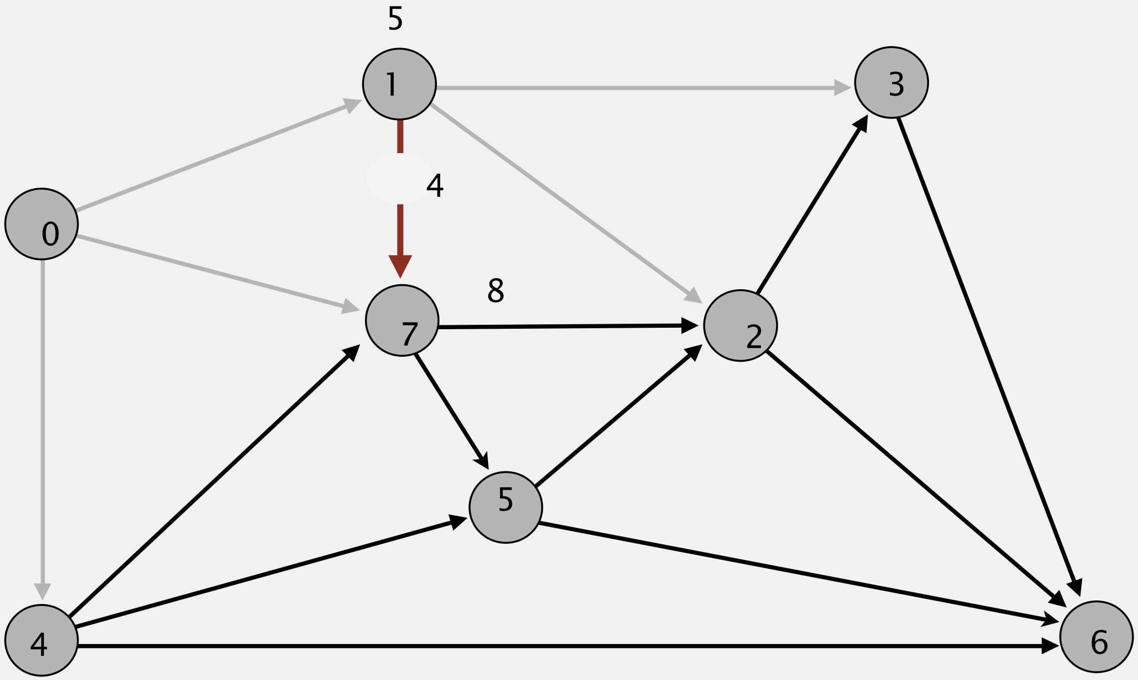
pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

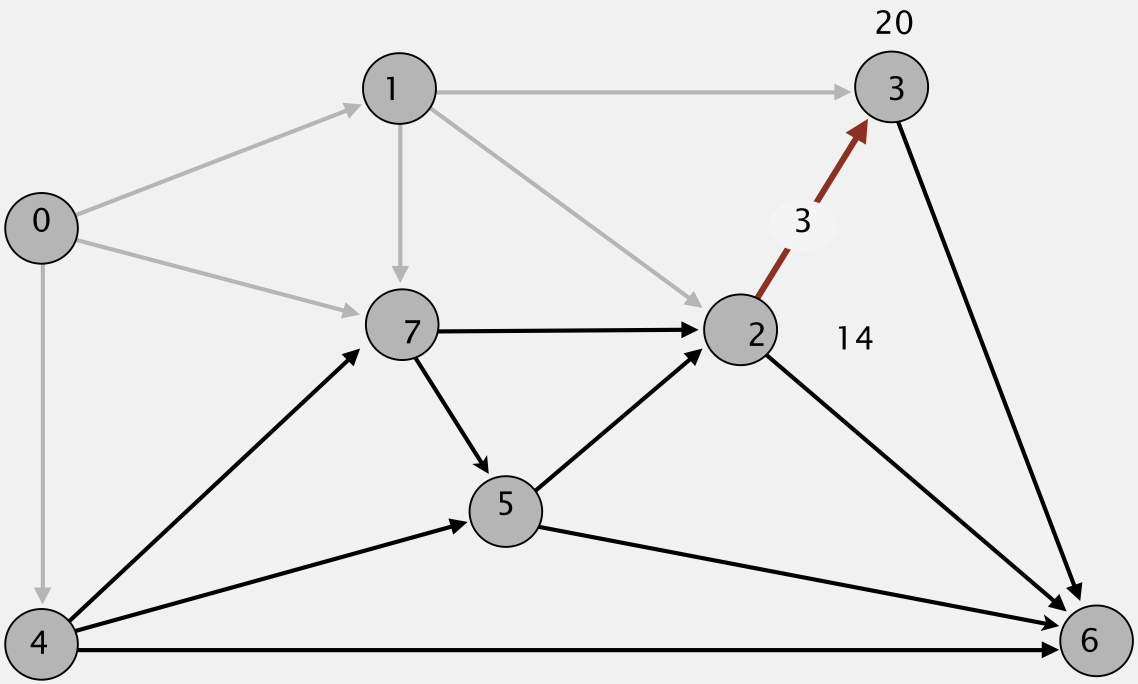
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

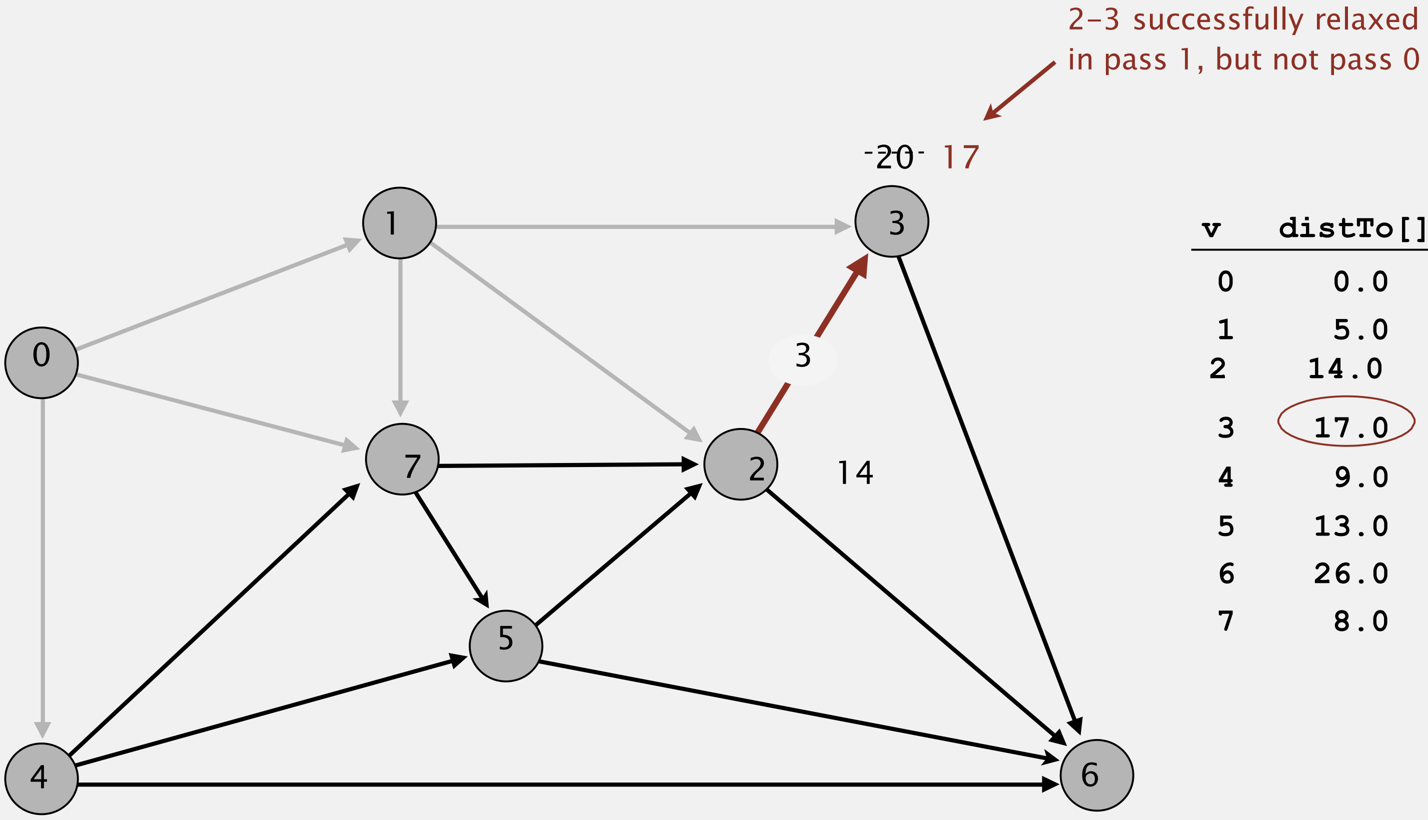
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

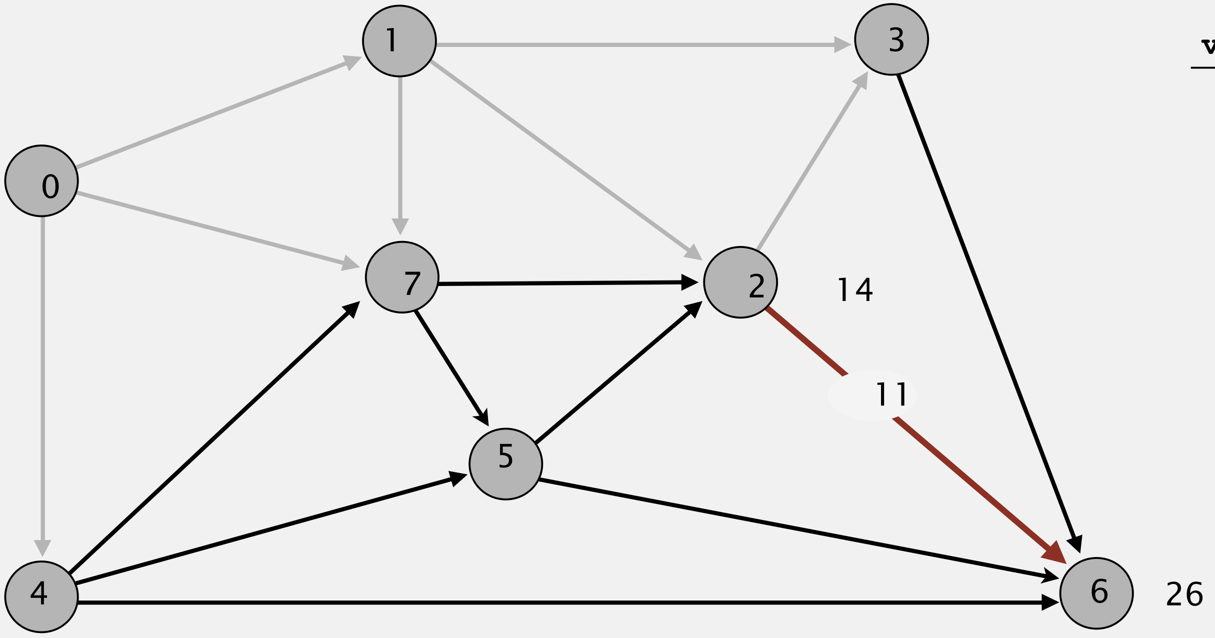
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

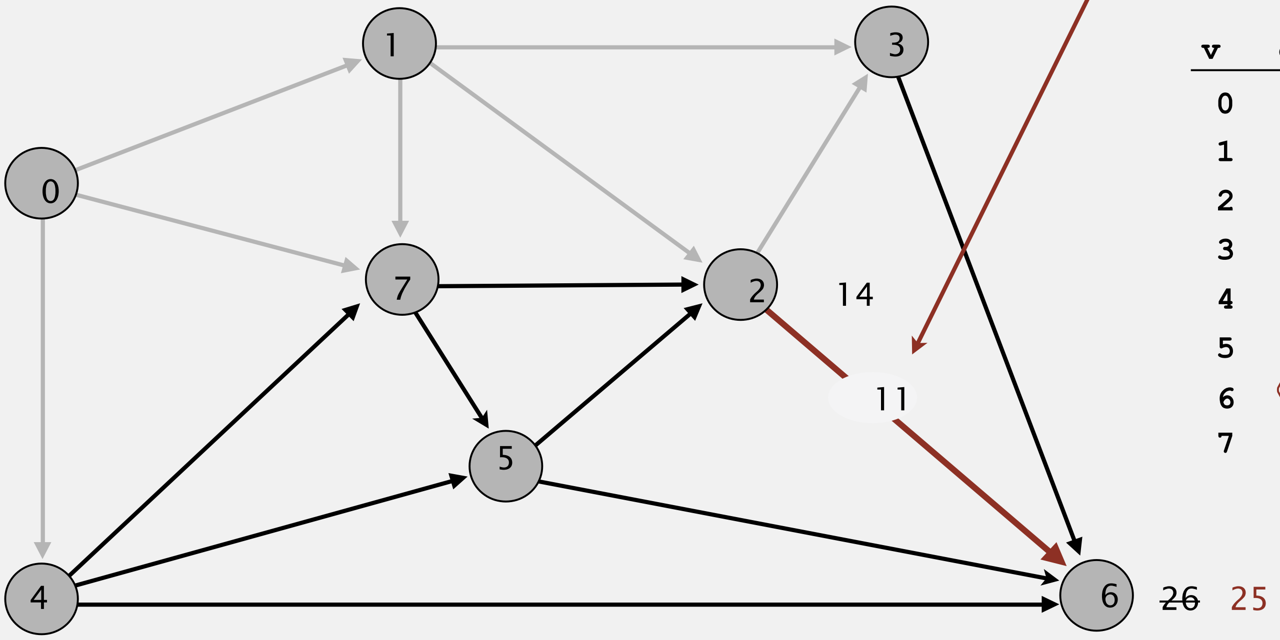
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



2–6 successfully relaxed
in pass 0 and pass 1

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

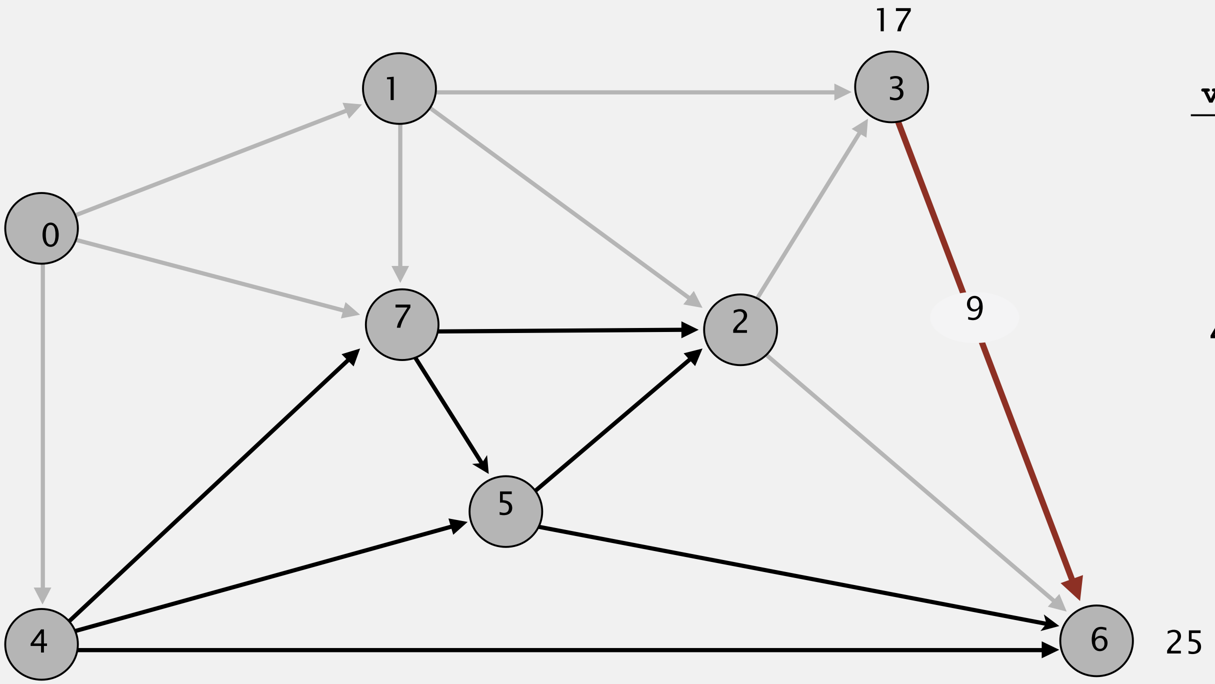
pass 1

`0→1  0→4  0→7  1→2  1→3  1→7  2→3  2→6  3→6  4→5  4→6  4→7  5→2  5→6  7→5  7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

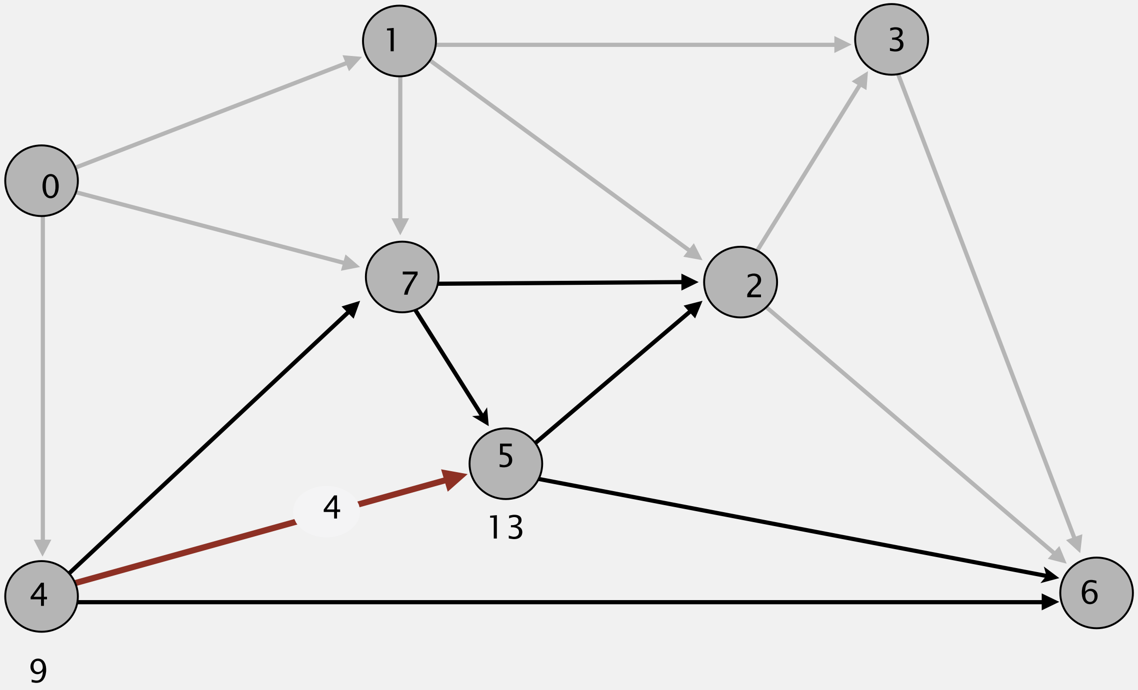
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

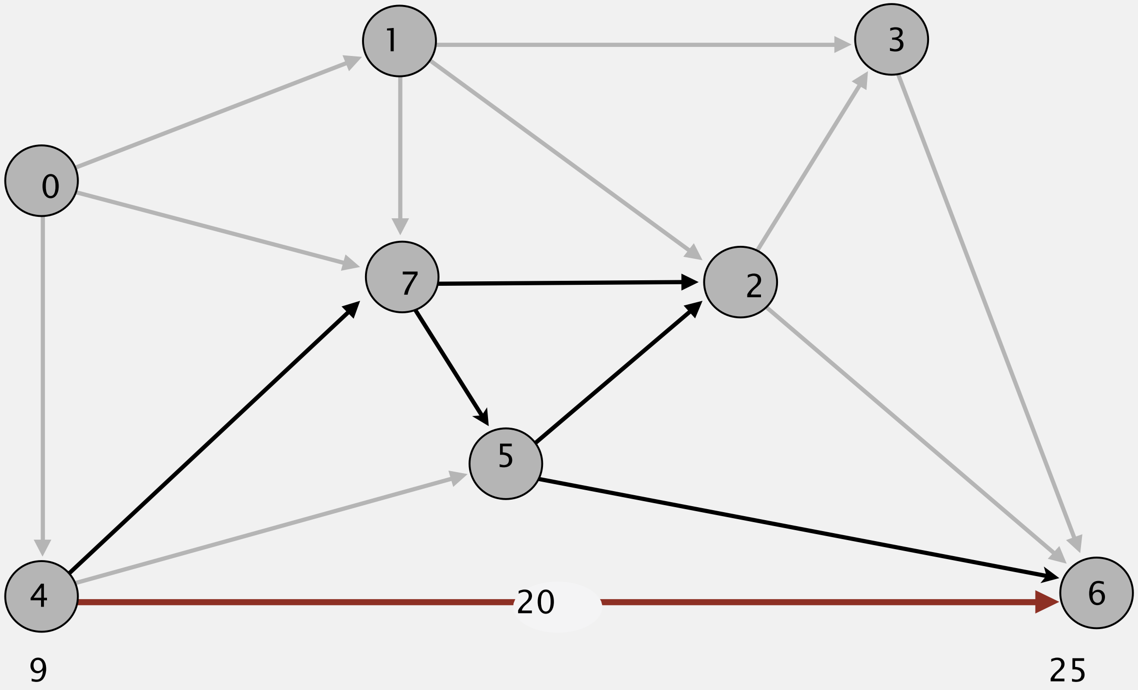
pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



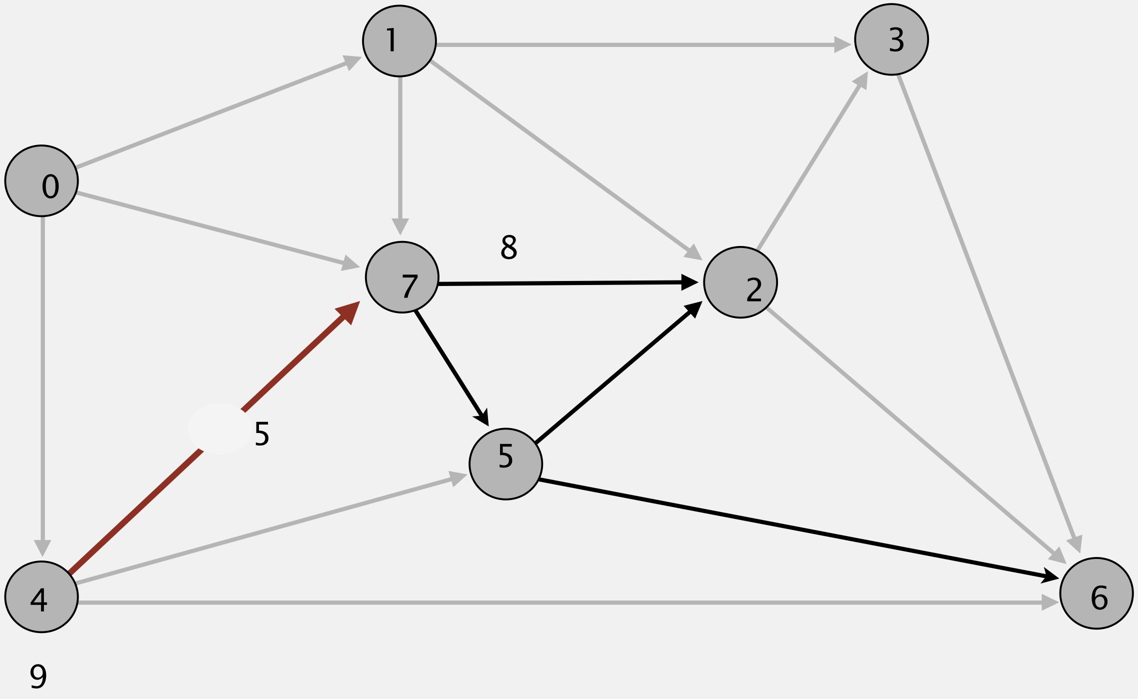
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



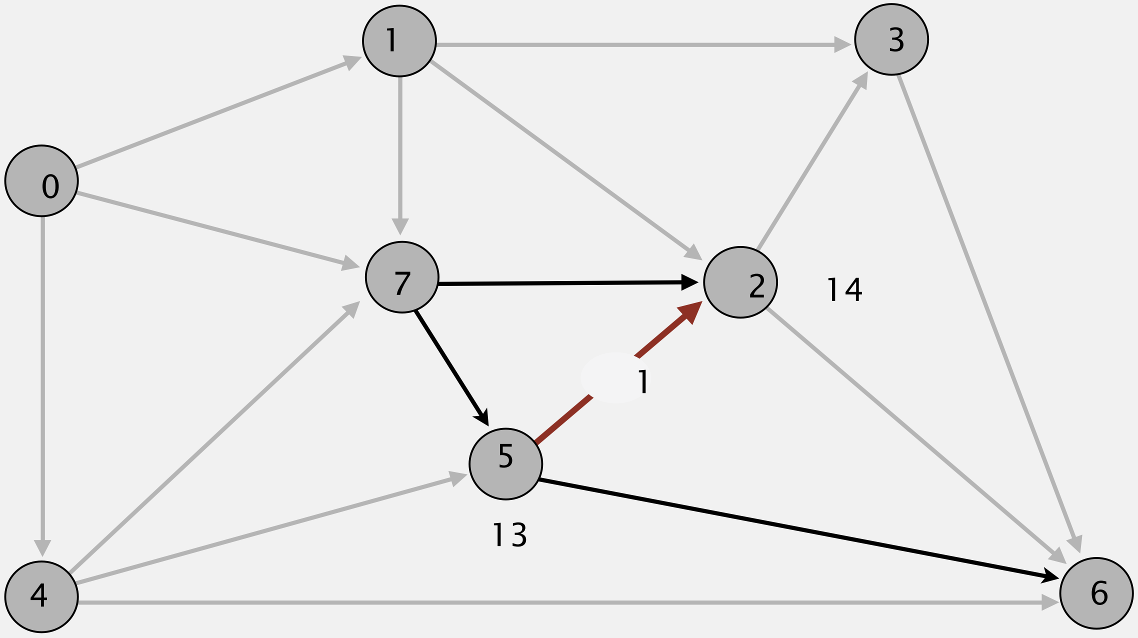
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



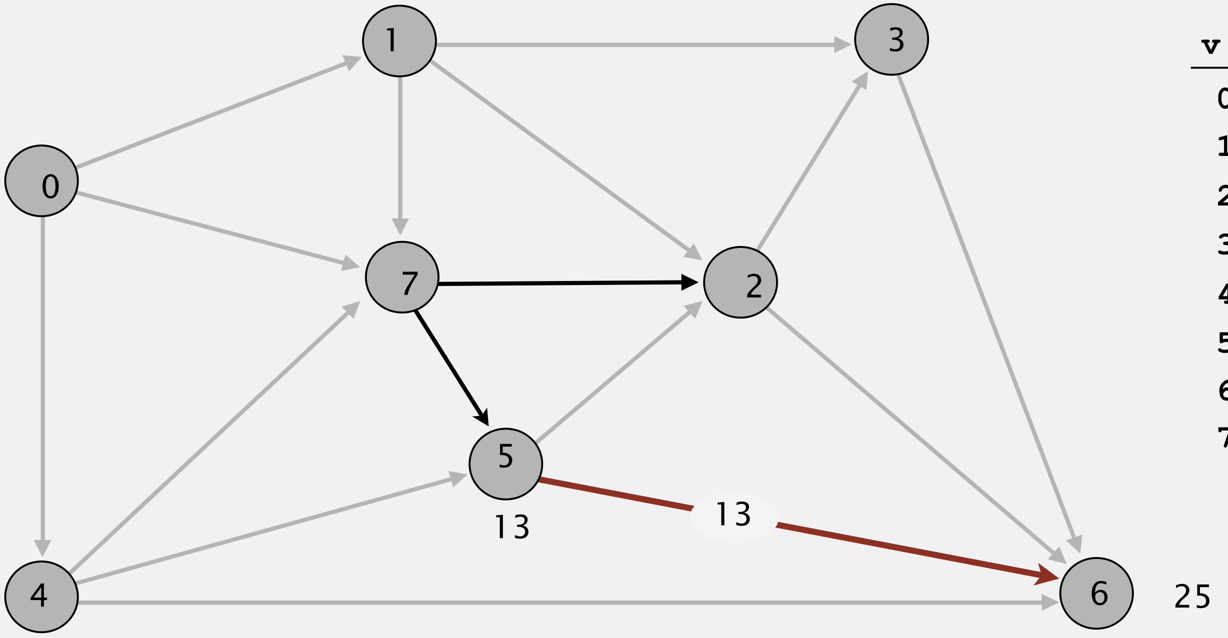
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

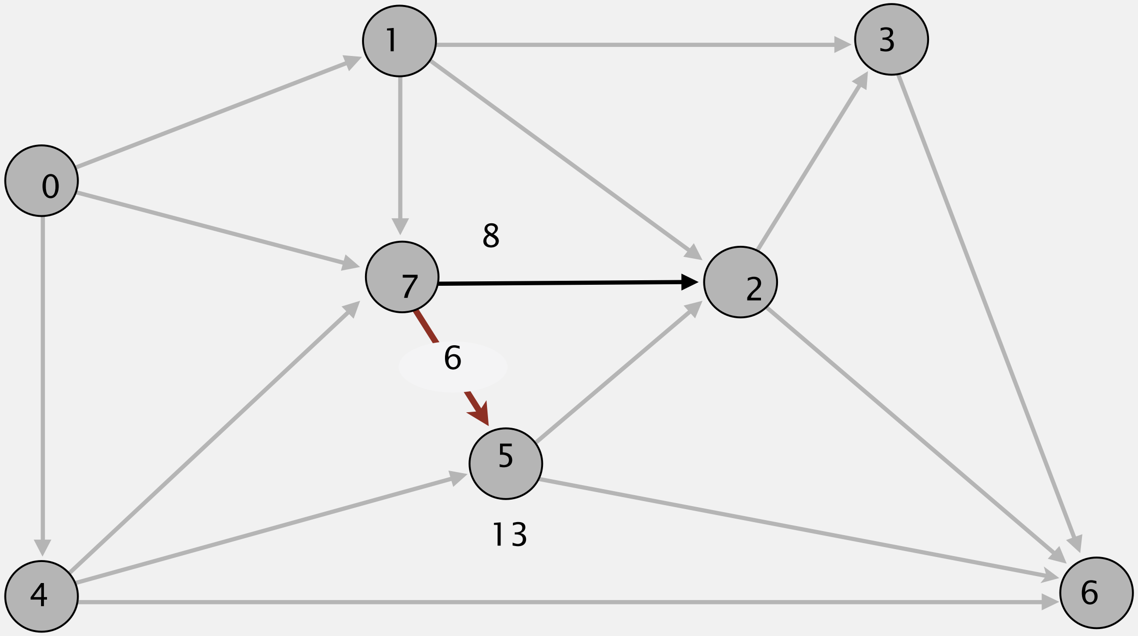
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

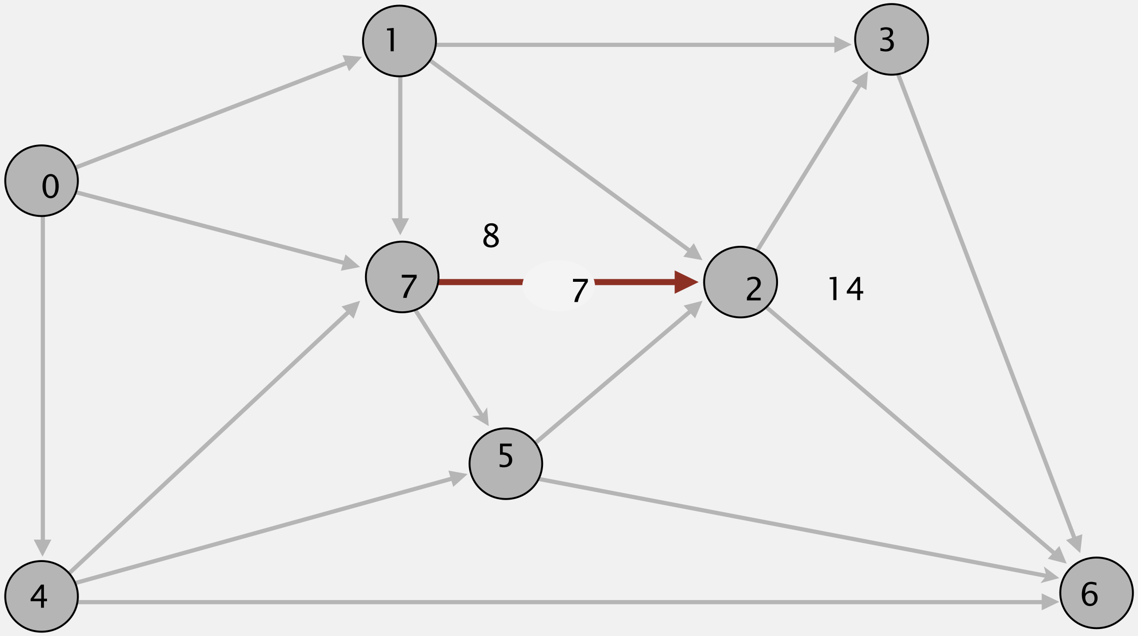
pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

Repeat V times: relax all E edges.



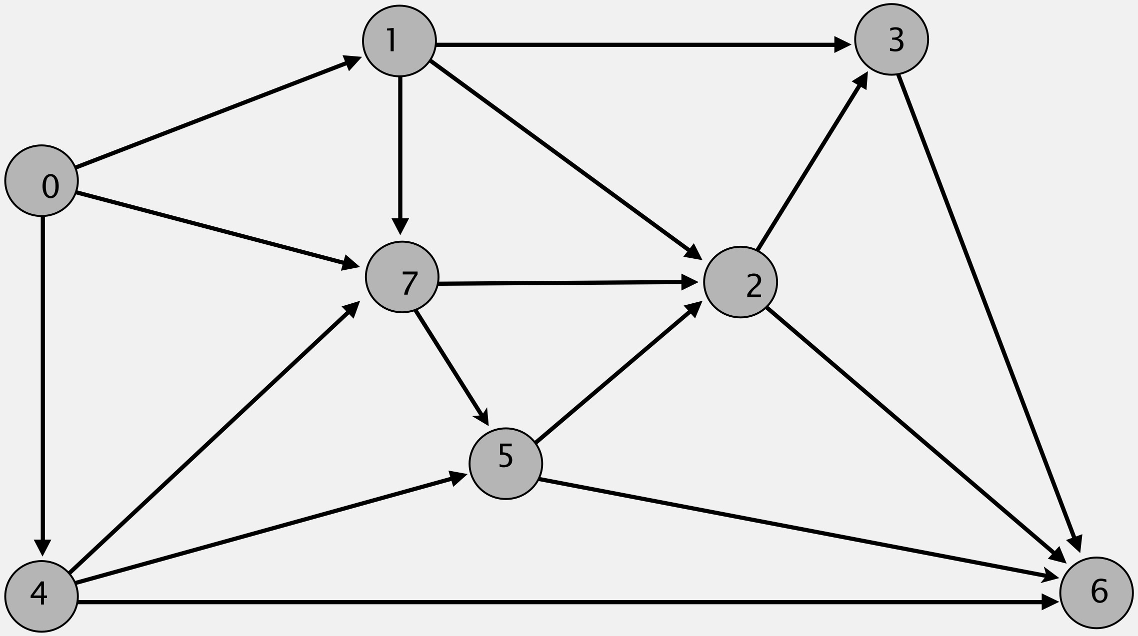
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

Bellman-Ford algorithm

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

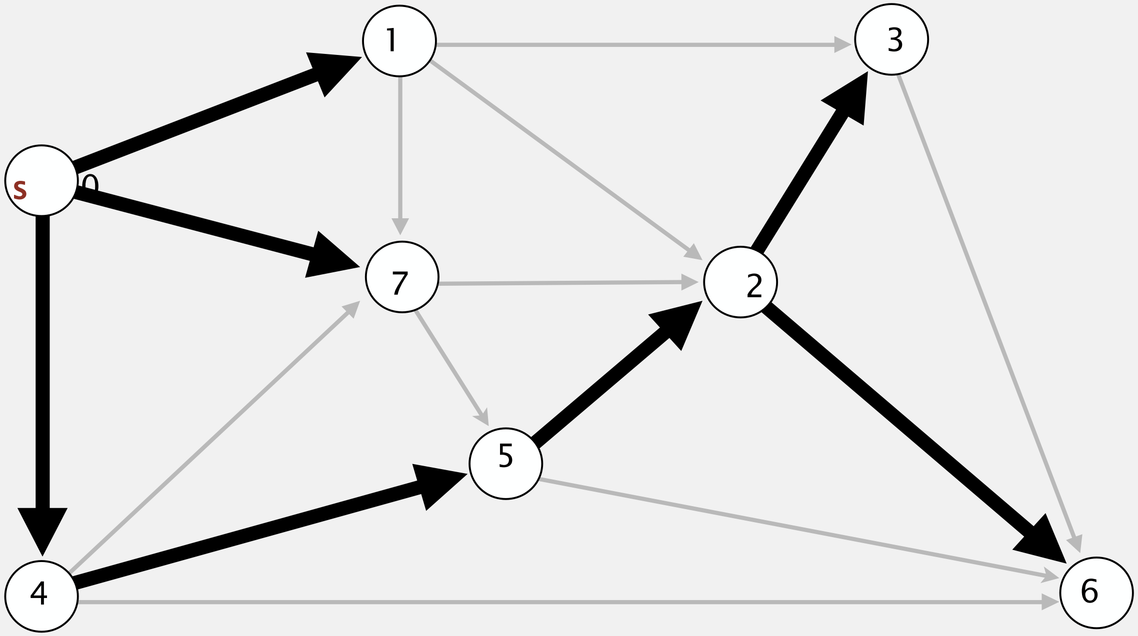
pass 2, 3, 4, ... (no further changes)

`0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2`

↑

Bellman-Ford algorithm

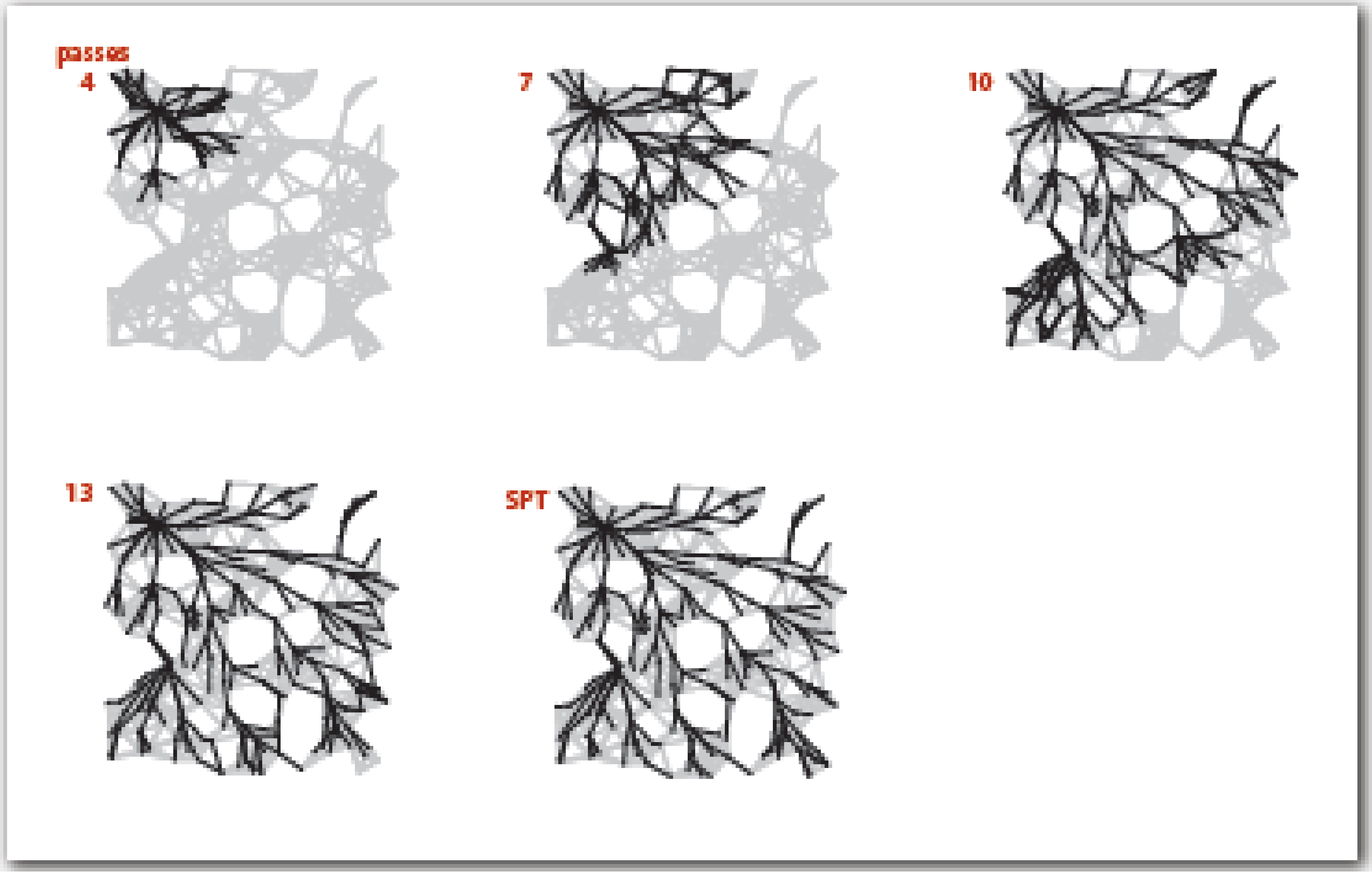
Repeat V times: relax all E edges.



shortest-paths tree from vertex s

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Bellman-Ford algorithm visualization



Bellman-Ford Algorithm: Correctness

- Shortest path will never visit the same vertex twice, so we know path consists of at most $V - 1$ edges
- Let $\delta(s, v_i)$ be the true shortest path that exists from s to v_i

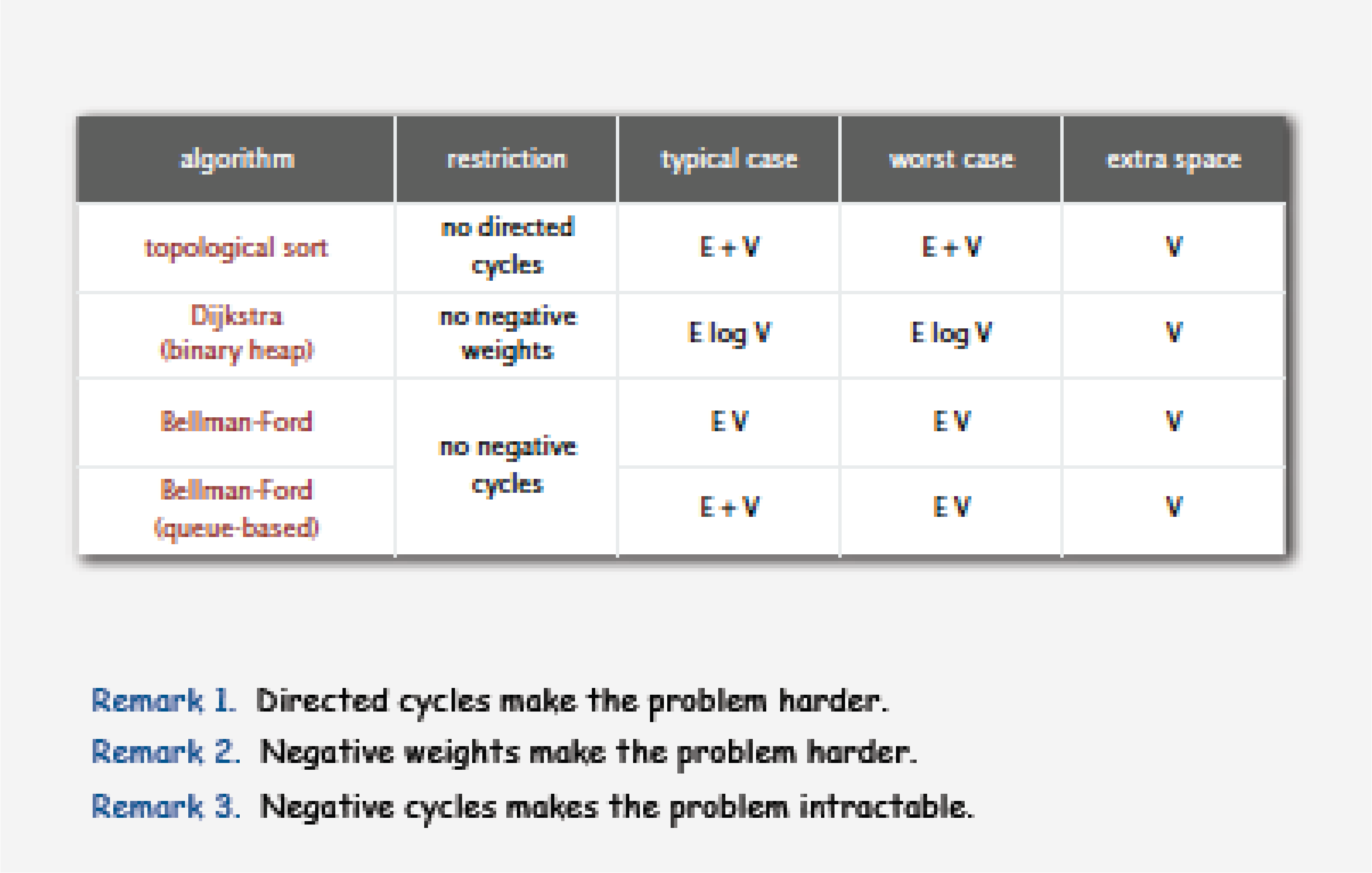
$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i).$$

- After the i^{th} pass of the “for-i” loop, $d[vi] \ = \ \delta(s, vi)$
- Proof by induction:
  - Base Case: $d[v_1] = d[s] = 0$
  - Prior to i^{th} pass: $d[v_{i-1}] = \delta(s, v_{i-1})$
  - After the i^{th} pass, we have done relaxation on the edge (v_{i-1}, vi)
  - Therefore, after the i^{th} pass we have:

$$d[v_i] \ \leq \ d[v_{i-1}] + w(v_{i-1}, v_i) \ = \ \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \ = \ \delta(s, v_i).$$

  - Recall from Dijkstra’s algorithm that $d[vi]$ is never less than $\delta(s, vi)$
  - Therefore, $d[vi] \ = \ \delta(s, vi)$

Single source shortest-paths implementation: cost summary



algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	E V	E V	V
Bellman-Ford (queue-based)		E + V	E V	V

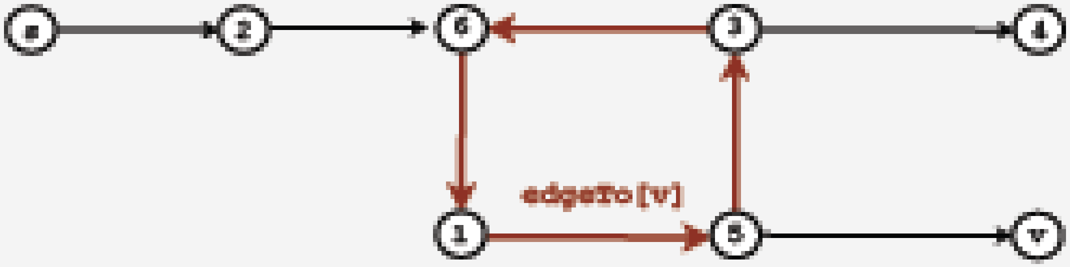
Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



Proposition. If any vertex v is updated in phase *V*, there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.