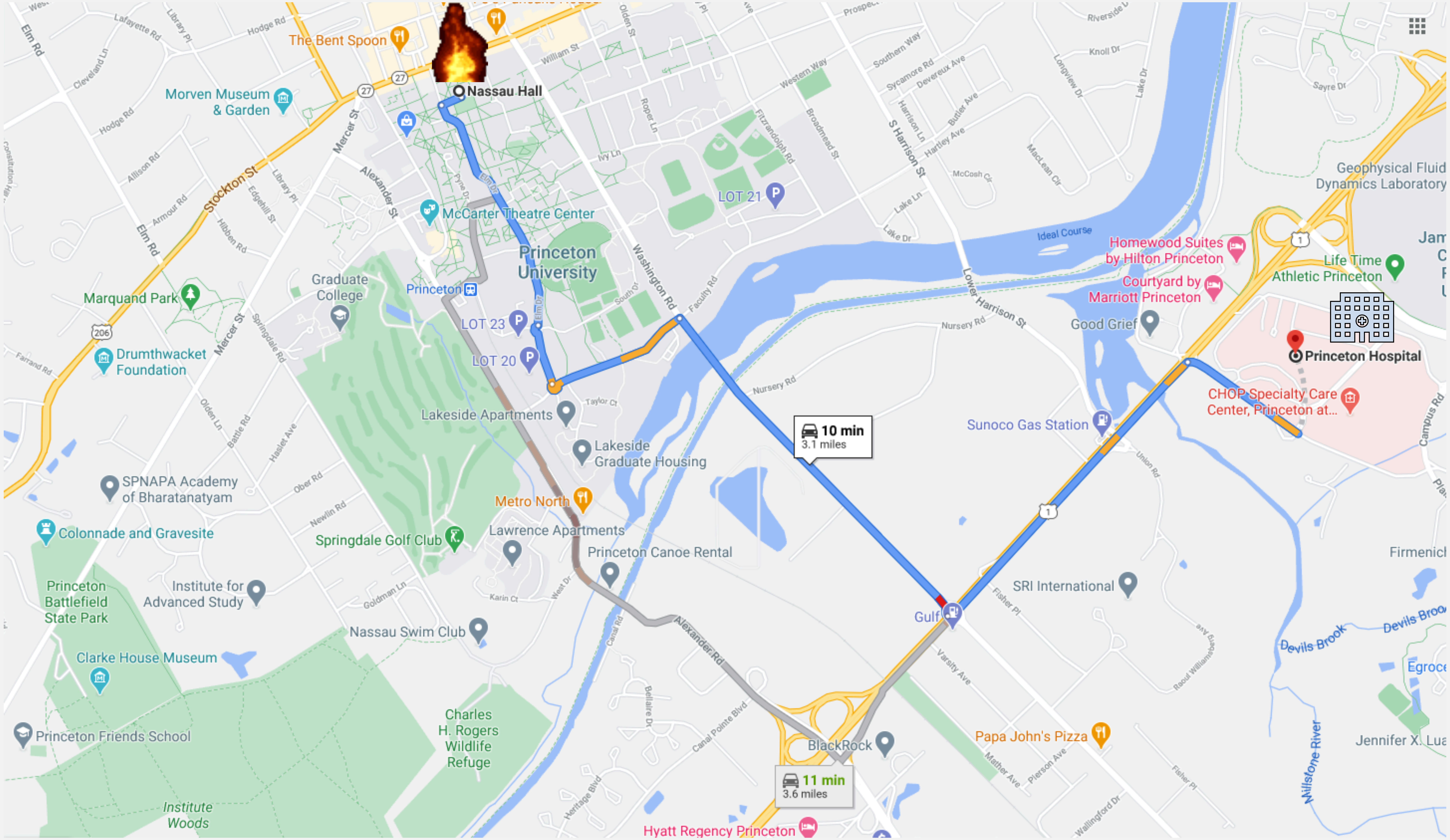
https://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Google maps

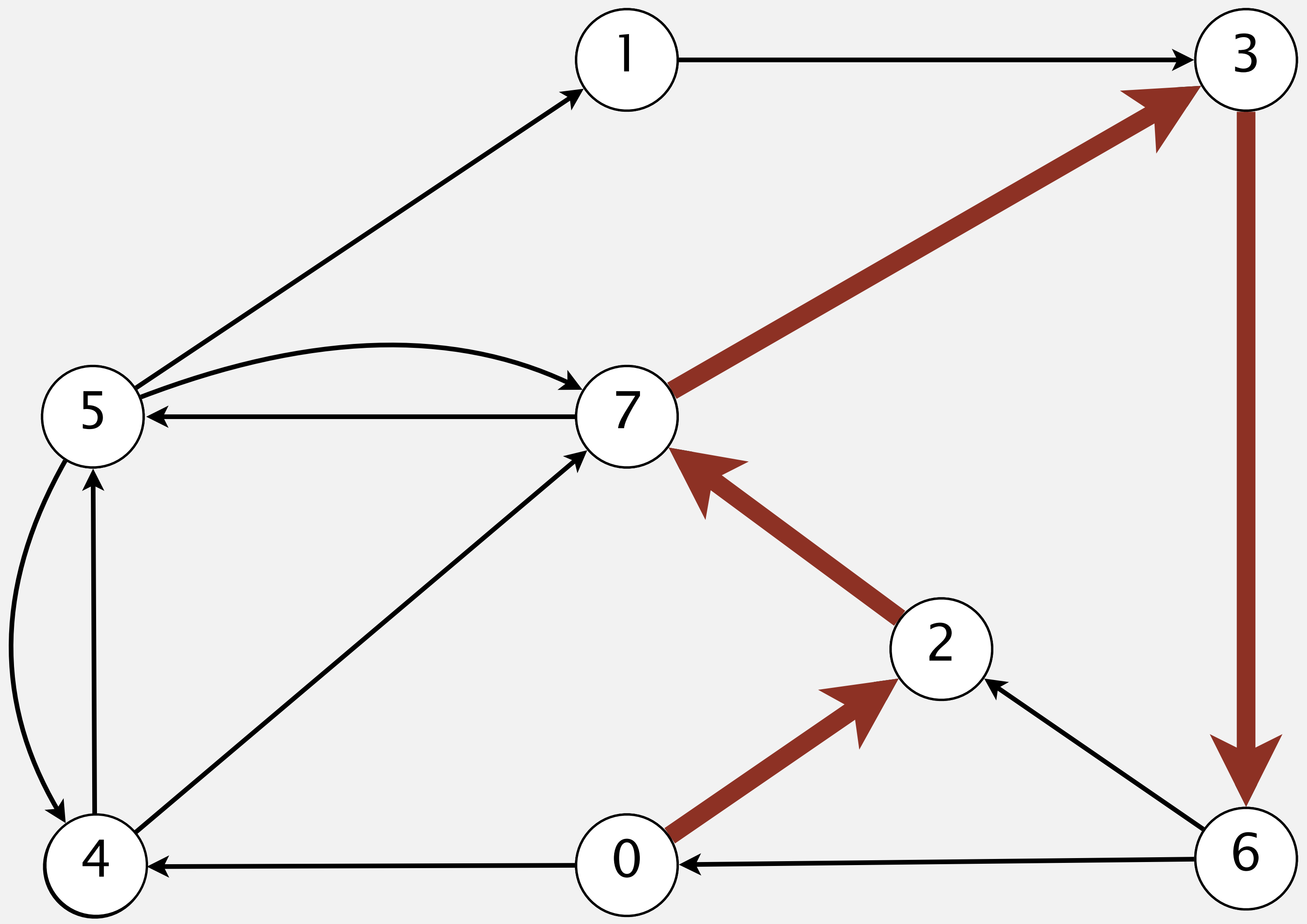

Nassau Hall
Princeton Hospital
The Bent Spoon
Morven Museum & Garden
McCarter Theatre Center
Princeton University
Graduate College
Princeton
LOT 23
LOT 20
LOT 21
Lakeside Apartments
Lakeside Graduate Housing
Metro North
Lawrence Apartments
Princeton Canoe Rental
10 min
3.1 miles
11 min
3.6 miles
Homewood Suites by Hilton Princeton
Courtyard by Marriott Princeton
Good Grief
Life Time Athletic Princeton
Geophysical Fluid Dynamics Laboratory
CHOP Specialty Care Center, Princeton at...
Sunoco Gas Station
SRI International
Gulf
BlackRock
Papa John's Pizza
Hyatt Regency Princeton
Marquand Park
Drumthwacket Foundation
SPNAPA Academy of Bharatanatyam
Colonnade and Gravesite
Springdale Golf Club
Princeton Battlefield State Park
Institute for Advanced Study
Nassau Swim Club
Clarke House Museum
Princeton Friends School
Charles H. Rogers Wildlife Refuge
Institute Woods
Firmenich
Devils Brook
Millstone River
Stockton St
Alexander Rd
Washington Rd
Lower Harrison St
S Harrison St
Mercer St
Nursery Rd

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from s to t.

edge-weighted digraph

```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



shortest path from 0 to 6

$0 \to 2 \to 7 \to 3 \to 6$

length of path = 1.51

$(0.26 + 0.34 + 0.39 + 0.52)$

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving. ← see Assignment 6
- Texture mapping.
- Robot navigation.
- Typesetting in $\TeX$.
- Currency exchange.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

https://en.wikipedia.org/wiki/Seam_carving

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

Shortest path variants

Which vertices?

- Single source: from one vertex s to every vertex.
- Single destination: from every vertex to one vertex t.
- Source–destination: from one vertex s to another vertex t.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Non-negative weights. ← we assume this in today's lecture (except as noted)
- Euclidean weights.
- Arbitrary weights.

Directed cycles?

- Prohibit.
- Allow.

implies that shortest path from s to v exists (and that $E \geq V - 1$)

Simplifying assumption. Each vertex is reachable from s.

Shortest paths: quiz 1

Which variant in car GPS? Hint: drivers make wrong turns occasionally.

A. Single source: from one vertex s to every vertex.

B. Single destination: from every vertex to one vertex t.

C. Source–destination: from one vertex s to another vertex t.

D. All pairs: between all pairs of vertices.

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Data structures for single-source shortest paths

Goal. Find a shortest path from s to every vertex.

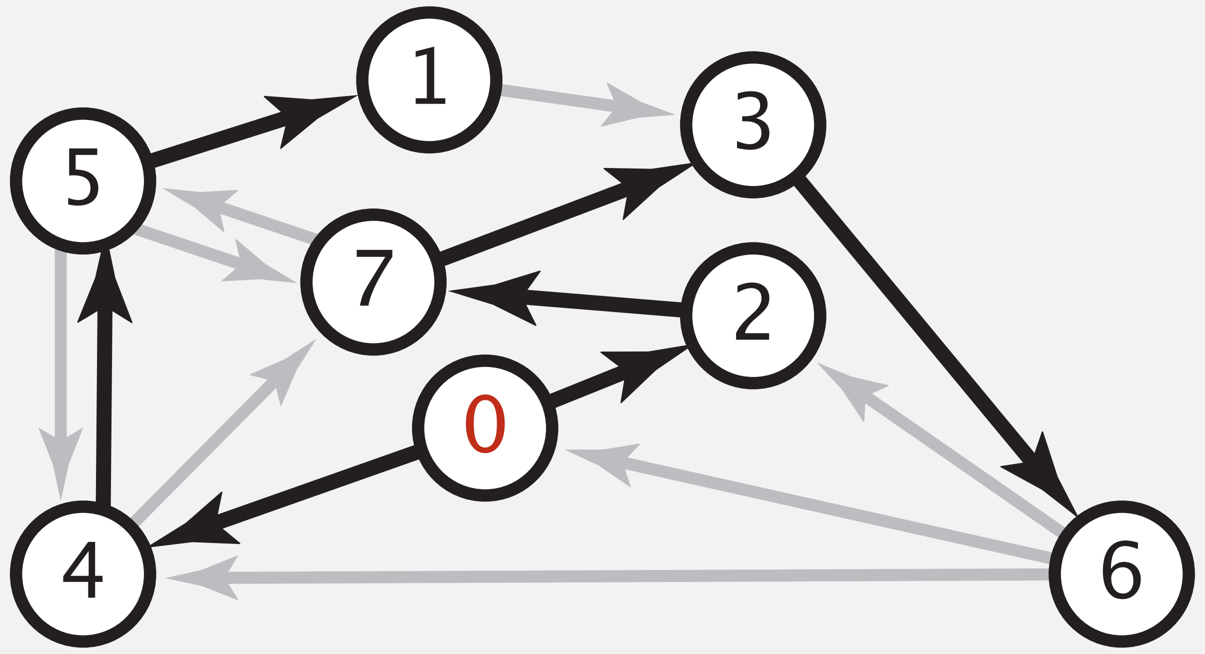
no repeated vertices $\Rightarrow \leq V - 1$ edges

Observation 1. There exists a shortest path from s to v that is simple.

Observation 2. A shortest-paths tree (SPT) solution exists. Why?

Consequence. Can represent a SPT with two vertex-indexed arrays:

- `distTo[v]` is length of a shortest path from s to v.
- `edgeTo[v]` is last edge on a shortest path from s to v.



shortest-paths tree from 0

	distTo[]	edgeTo[]
0	0	null
1	1.05	5->1 0.32
2	0.26	0->2 0.26
3	0.97	7->3 0.37
4	0.38	0->4 0.38
5	0.73	4->5 0.35
6	1.49	3->6 0.52
7	0.60	2->7 0.34

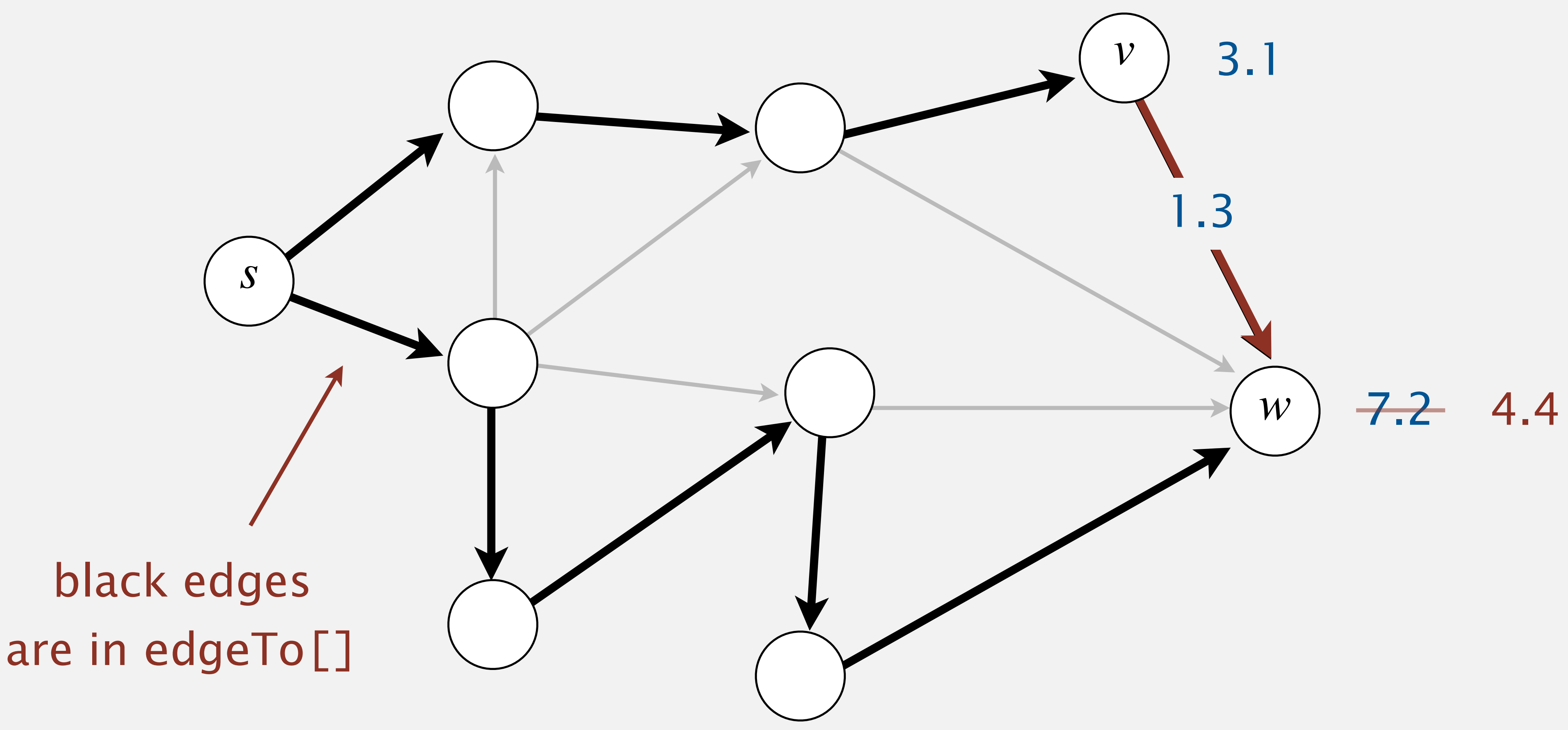
parent-link representation

Edge relaxation

Relax edge $e = v \to w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \to w$ yields shorter path from s to w, via v, update `distTo[w]` and `edgeTo[w]`.

relax edge e = v→w



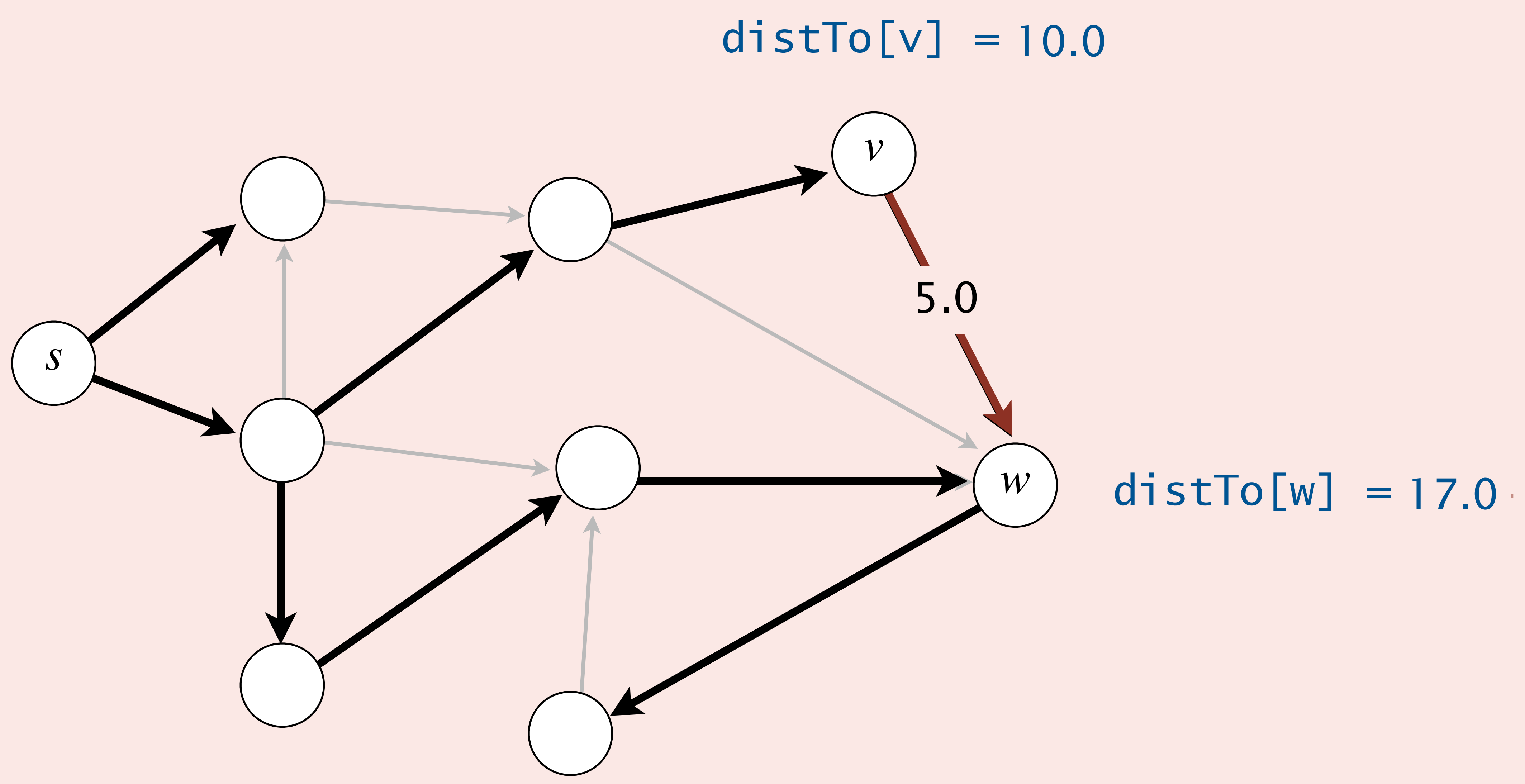
What are the values of `distTo[v]` and `distTo[w]` after relaxing $e = v \rightarrow w$?

A. 10.0 and 15.0

B. 10.0 and 17.0

C. 12.0 and 15.0

D. 12.0 and 17.0



Framework for shortest-paths algorithm

Generic algorithm (to compute a SPT from s)

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat until distTo[v] values converge:

- **Relax any edge.**

Key properties. Throughout the generic algorithm,

- `distTo[v]` is either infinity or the length of a (simple) path from s to v.
- `distTo[v]` does not increase.

Framework for shortest-paths algorithm

Generic algorithm (to compute a SPT from s)

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat until distTo[v] values converge:

- **Relax any edge.**

Efficient implementations.

- Which edge to relax next?
- How many edge relaxations needed to guarantee convergence?

Ex 1. Bellman–Ford algorithm.

Ex 2. Dijkstra's algorithm.

Ex 3. Topological sort algorithm.

https://algs4.cs.princeton.edu

4.4 Shortest Paths

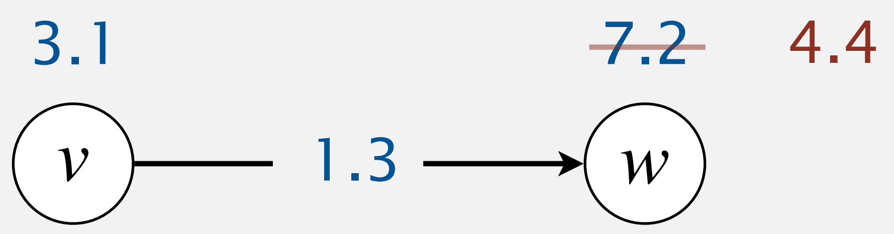
- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Weighted directed edge API

`public class DirectedEdge`		
	`DirectedEdge(int v, int w, double weight)`	*weighted edge v→w*
`int`	`from()`	*vertex v*
`int`	`to()`	*vertex w*
`double`	`weight()`	*weight of this edge*

Relaxing an edge $e = v \to w$.

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
   }
}
```



Weighted directed edge: implementation in Java

API. Similar to Edge for undirected graphs, but a bit simpler.

```
public class DirectedEdge
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
     this.v = v;
     this.w = w;
     this.weight = weight;
   }

   public int from()
   {  return v;  }

   public int to()
   {  return w; }

   public double weight()
   {  return weight;  }

}
```

from() and to() replace either() and other()

Edge-weighted digraph API

API. Same as `EdgeWeightedGraph` except with `DirectedEdge` objects.

public class EdgeWeightedDigraph		
	`EdgeWeightedDigraph(int V)`	*edge-weighted digraph with V vertices*
`void`	`addEdge(DirectedEdge e)`	*add weighted directed edge e*
`Iterable<DirectedEdge>`	`adj(int v)`	*edges incident from v*
`int`	`V()`	*number of vertices*
	⋮	⋮

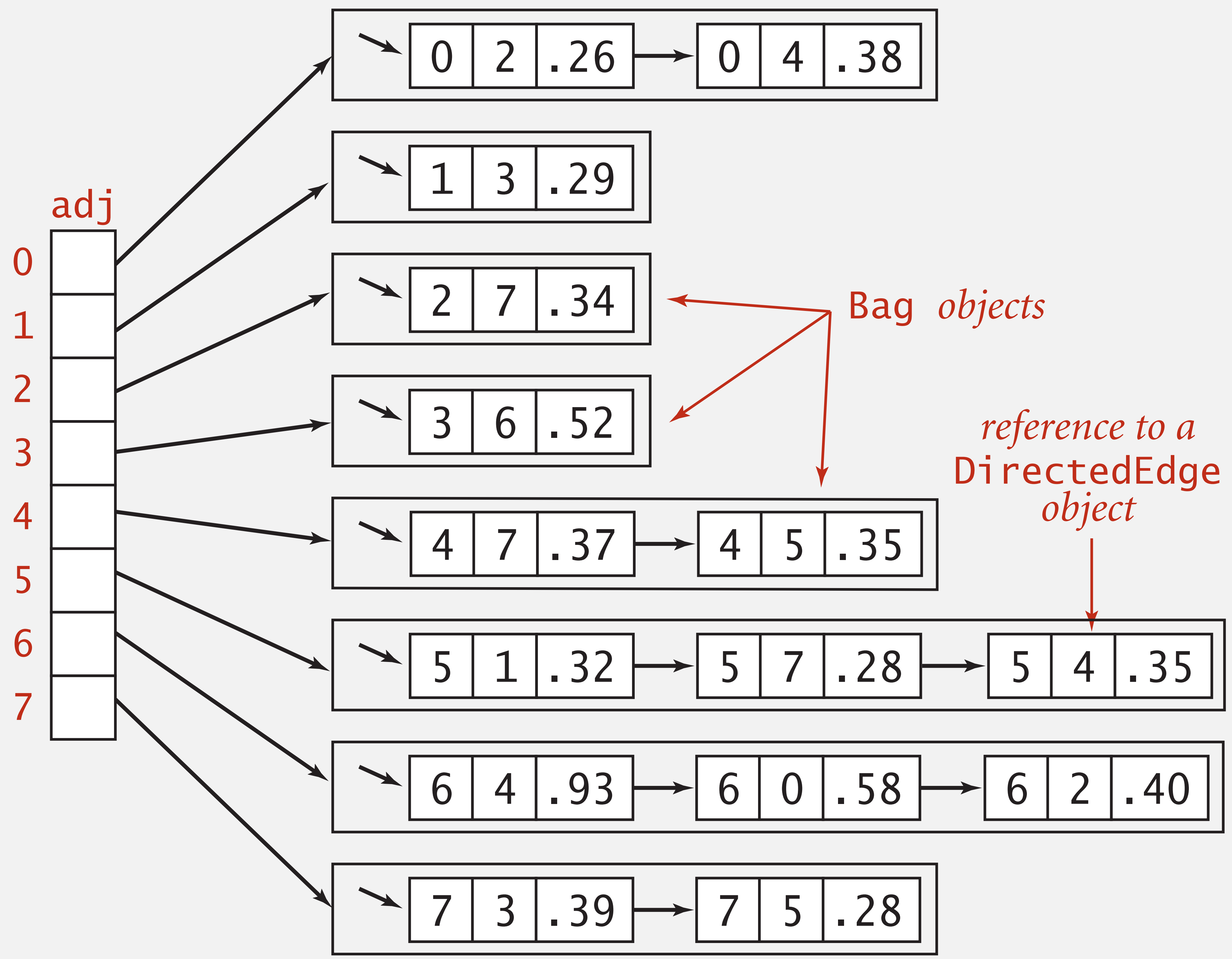
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V → 8

E → 15

```
4 5  0.35
5 4  0.35
4 7  0.37
5 7  0.28
7 5  0.28
5 1  0.32
0 4  0.38
0 2  0.26
7 3  0.39
1 3  0.29
2 7  0.34
6 2  0.40
3 6  0.52
6 0  0.58
6 4  0.93
```



Edge-weighted digraph: adjacency-lists implementation in Java

Implementation. Almost identical to `EdgeWeightedGraph`.

```
public class EdgeWeightedDigraph
{
   private final int V;
   private final Bag<DirectedEdge>[] adj;

   public EdgeWeightedDigraph(int V)
   {
      this.V = V;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<>();
   }

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

   public Iterable<DirectedEdge> adj(int v)
   {  return adj[v];  }
}
```

add edge $e = v \rightarrow w$ to only v's adjacency list

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

public class SP		
	`SP(EdgeWeightedDigraph G, int s)`	*shortest paths from s in digraph G*
`double`	`distTo(int v)`	*length of shortest path from s to v*
`Iterable <DirectedEdge>`	`pathTo(int v)`	*shortest path from s to v*
`boolean`	`hasPathTo(int v)`	*is there a path from s to v?*

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

Bellman–Ford algorithm

Bellman–Ford algorithm

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

distTo[s] = 0.

Repeat V–1 times:

– Relax each edge.

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
   }
}
```

```
for (int i = 1; i < G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

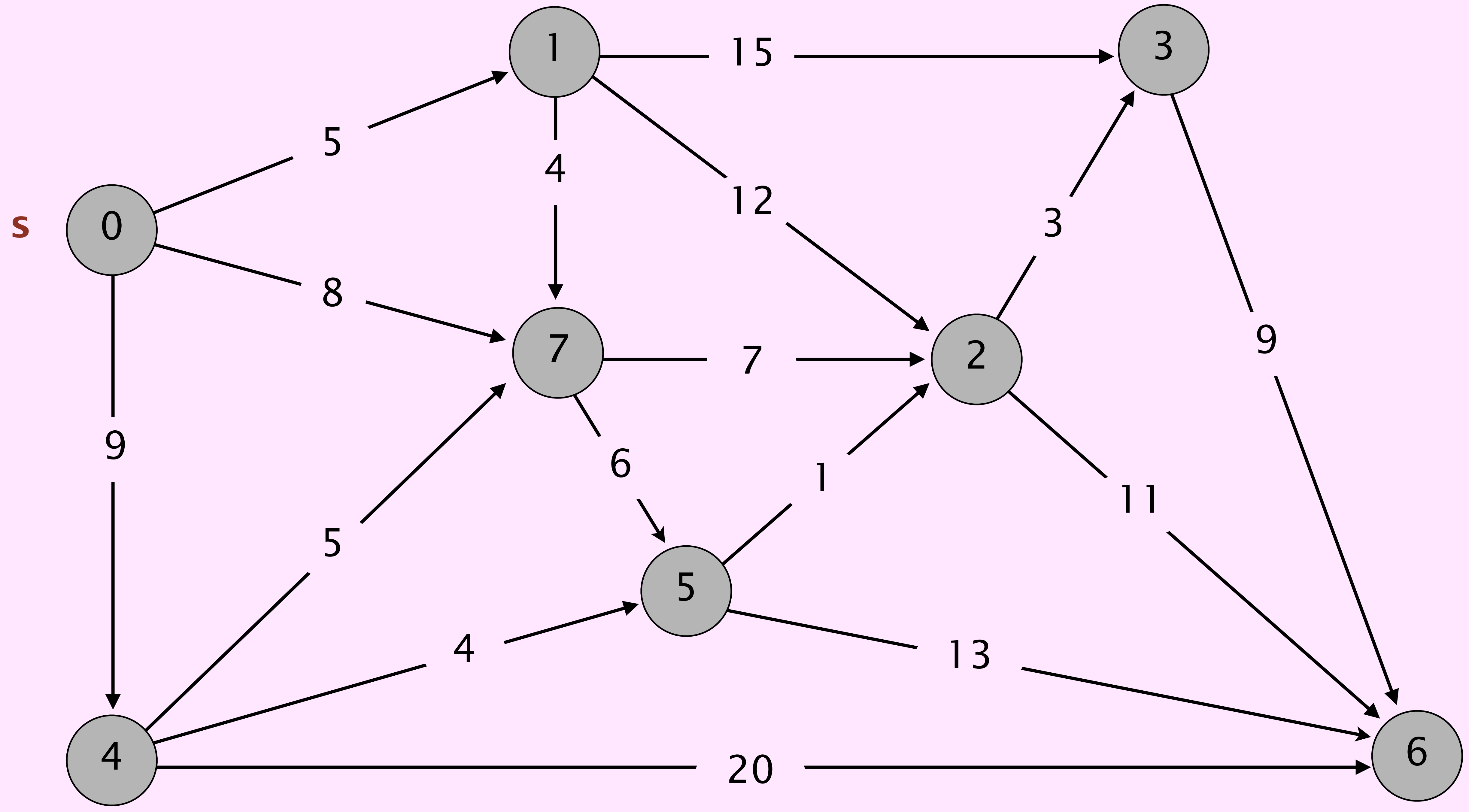
pass i (relax each edge once)

number of calls to relax() in pass i = $outdegree(0) + outdegree(1) + outdegree(2) + \ldots = E$

Running time. Algorithm takes $\Theta(E\,V)$ time and uses $\Theta(V)$ extra space.

Bellman–Ford algorithm demo

Repeat $V-1$ times: relax all E edges.

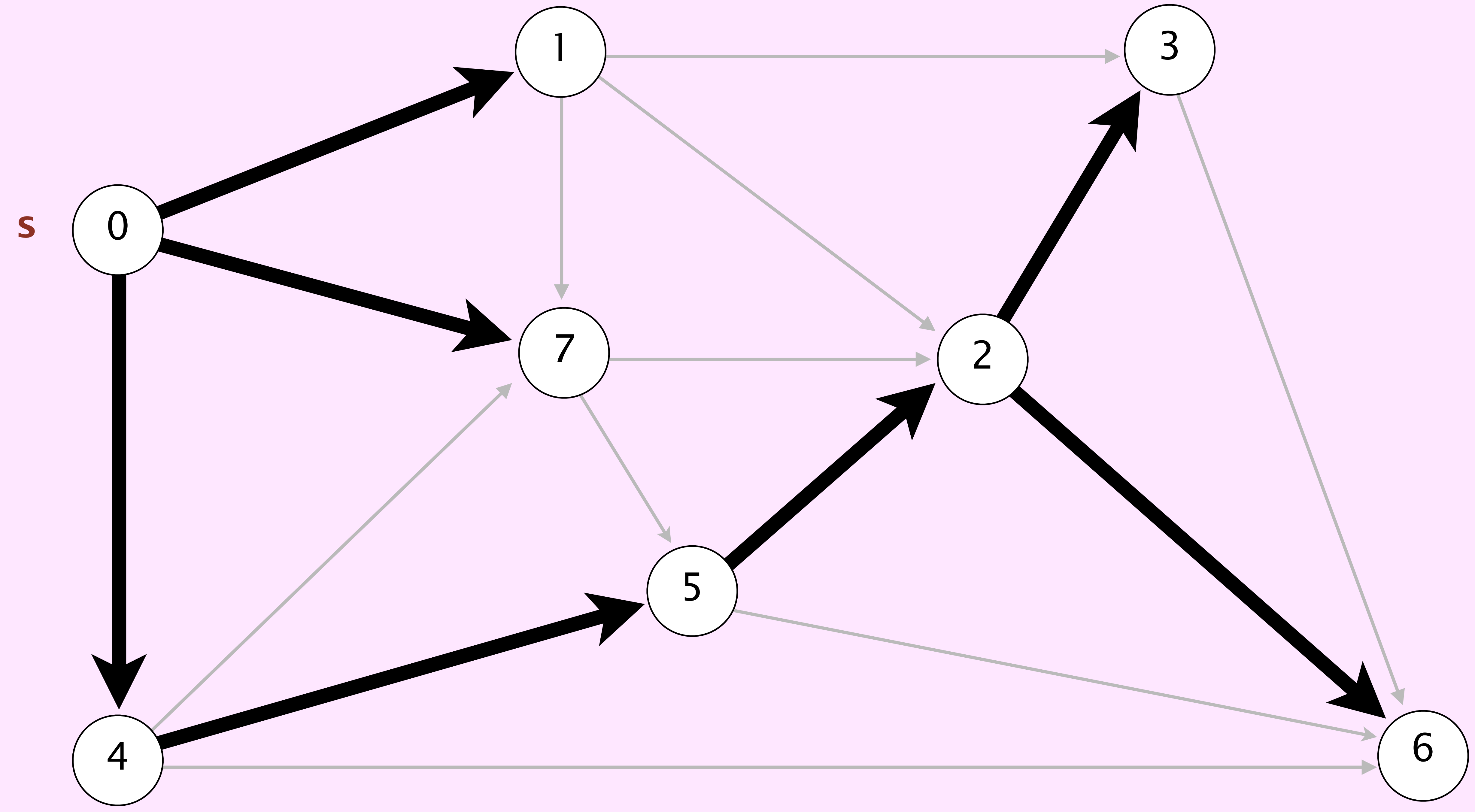


an edge-weighted digraph

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 demo

Repeat $V - 1$ times: relax all E edges.



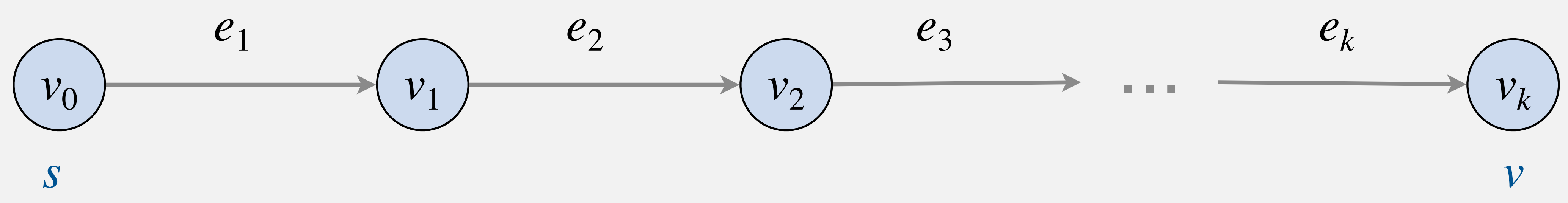
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Bellman–Ford algorithm: correctness proof

Proposition. Let $s = v_0 \to v_1 \to \ldots \to v_k = v$ be any path from s to v containing k edges.

Then, after pass k, `distTo[`v_k`]` $\leq weight(e_1) + weight(e_2) + \cdots + weight(e_k)$.



Pf. [by induction on number of passes i]

- Base case: initially, `distTo[`v_0`]` ≤ 0.
- Inductive hypothesis: after pass i, `distTo[`v_i`]` $\leq weight(e_1) + weight(e_2) + \cdots + weight(e_i)$.
- This inequality continues to hold because `distTo[`v_i`]` cannot increase.
- Immediately after relaxing edge e_{i+1} in pass $i+1$, we have

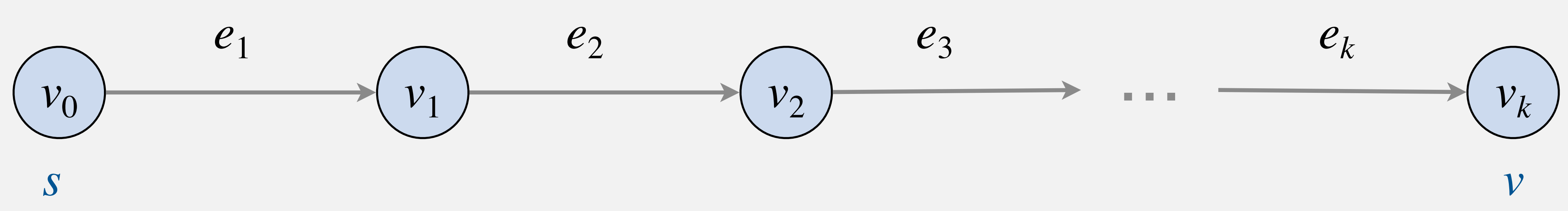
`distTo[`v_{i+1}`]` $\leq$ `distTo[`v_i`]` $+ \; weight(e_{i+1})$ ← edge relaxation

$\leq weight(e_1) + weight(e_2) + \cdots + weight(e_i) + weight(e_{i+1})$. ← inductive hypothesis

- This inequality continues to hold because `distTo[`v_{i+1}`]` does not increase. ▪

Bellman–Ford algorithm: correctness proof

Proposition. Let $s = v_0 \to v_1 \to \ldots \to v_k = v$ be any path from s to v containing k edges. Then, after pass k, `distTo[`v_k`]` $\leq weight(e_1) + weight(e_2) + \cdots + weight(e_k)$.



Corollary. Bellman–Ford computes shortest path distances.

Pf. [apply Proposition to a shortest path from s to v]

- There exists a shortest path P^* from s to v with $k \leq V - 1$ edges.
- From Proposition, `distTo[`v`]` $\leq length(P^*)$. ⟵ Bellman–Ford runs for V–1 passes
- Since `distTo[`v`]` is the length of some path from s to v, `distTo[`v`]` $= length(P^*)$. ▪

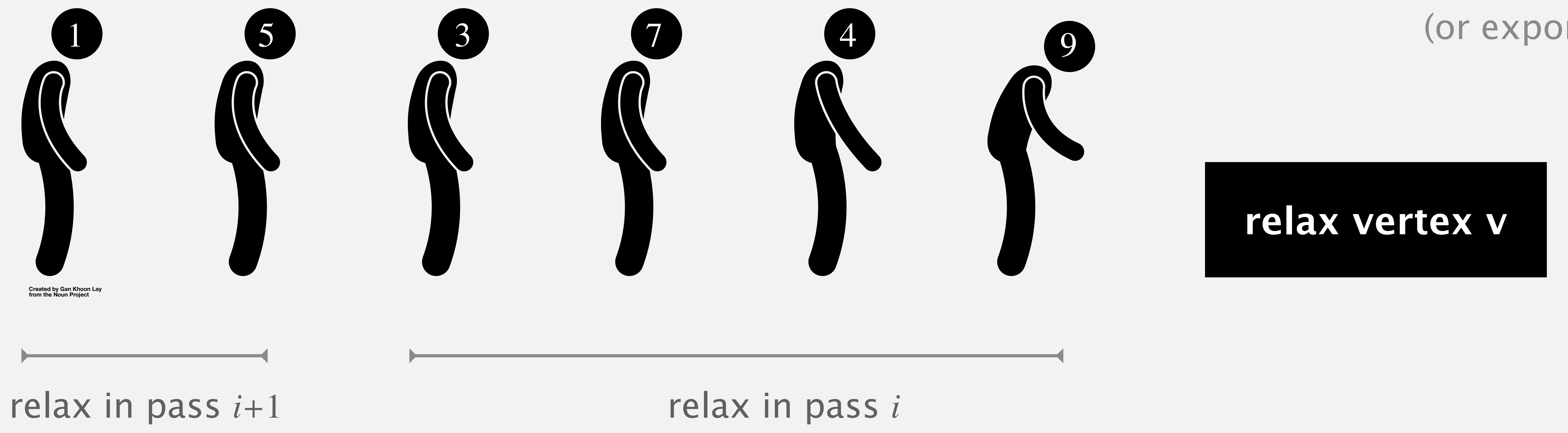
Bellman–Ford algorithm: practical improvement

Observation. If `distTo[v]` does not change during pass i, not necessary to relax any edges incident from v in pass $i + 1$.

Queue-based implementation of Bellman–Ford.

- Perform vertex relaxations. ← relax all edges incident from v
- Maintain queue of vertices whose `distTo[]` values changed since it was last relaxed.

must ensure each vertex is on queue at most once (or exponential blowup!)



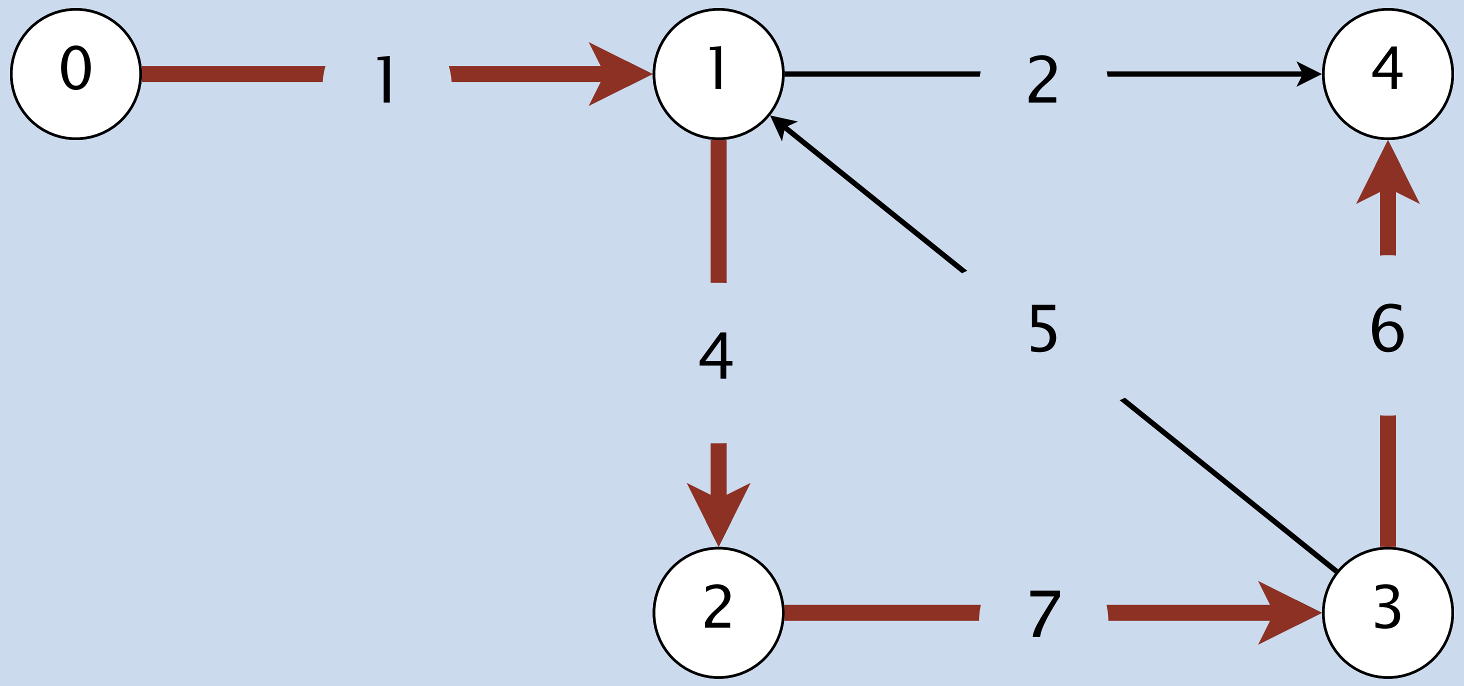
Impact.

- In the worst case, the running time is still $\Theta(E\,V)$.
- But much faster in practice on typical inputs.

Longest Path

Problem. Given a digraph G with positive edge weights and vertex s, find a longest simple path from s to every other vertex.

Goal. Design algorithm that takes $\Theta(E\,V)$ time in the worst case.

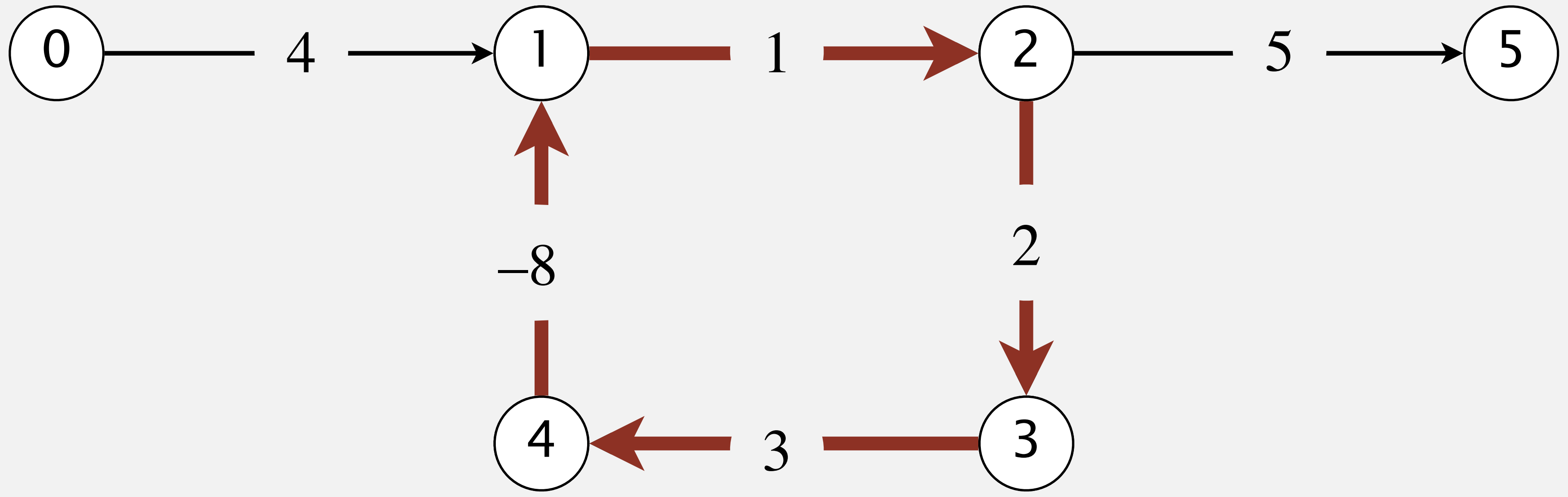


longest simple path from 0 to 4: 0→1→2→3→4

Bellman–Ford algorithm: negative weights

Remark. The Bellman–Ford algorithm works even if some weights are negative, provided there are no negative cycles.

Negative cycle. A directed cycle whose length is negative.



length of negative cycle = 1 + 2 +3 + –8 = –2

Negative cycles and shortest paths. Length of path can be made arbitrarily negative by using negative cycle.

$$0 \to 1 \to 2 \to 3 \to 4 \to \cdots \to 1 \to 2 \to 3 \to 4 \to 1 \to 2 \to 5$$

https://algs4.cs.princeton.edu

4.4 Shortest Paths

- properties
- APIs
- Bellman–Ford algorithm
- *Dijkstra's algorithm*

Edsger W. Dijkstra: select quotes

Dijkstra's algorithm

Dijkstra's algorithm

For each vertex v: distTo[v] = ∞.

For each vertex v: edgeTo[v] = null.

T = ∅.

distTo[s] = 0.

Repeat until all vertices are marked:

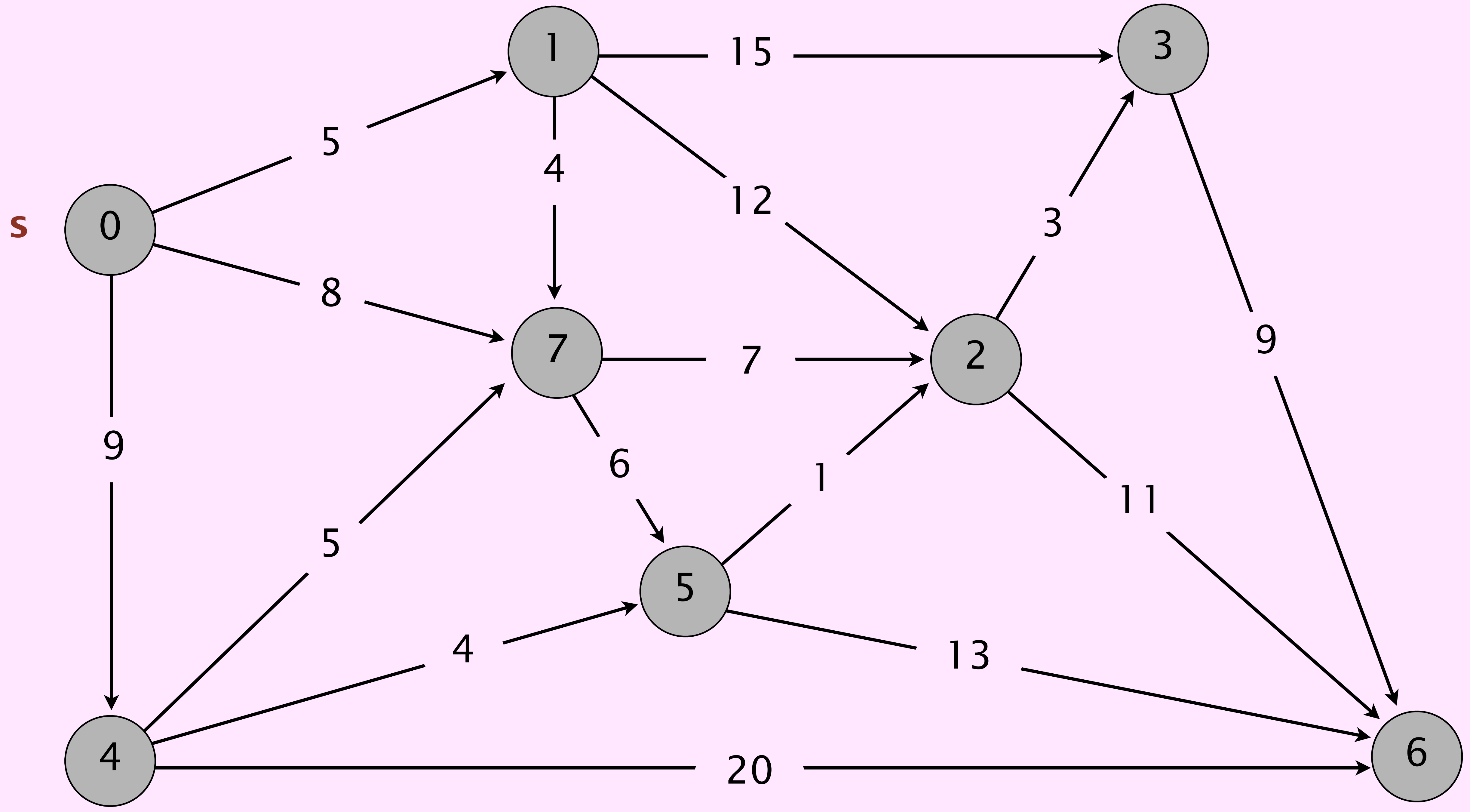
- Select unmarked vertex v with the smallest distTo[] value.
- Mark v.
- Relax each edge incident from v.

Key difference with Bellman–Ford. Each edge gets relaxed exactly once!

Dijkstra's algorithm demo

Repeat until all vertices are marked:

- Select unmarked vertex v with the smallest `distTo[]` value.
- Mark v and relax all edges incident from v.



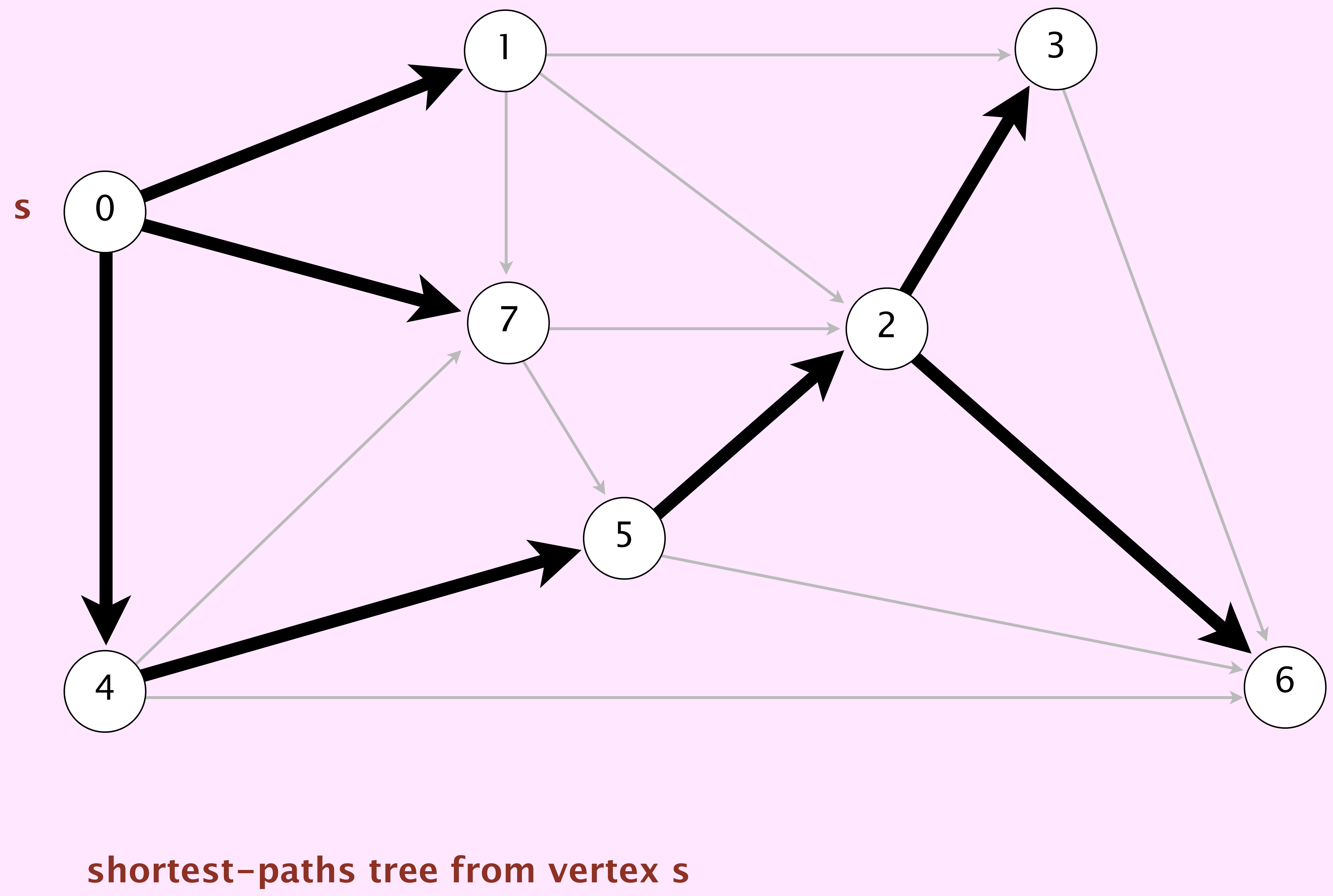
an edge-weighted digraph

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

Dijkstra's algorithm demo

Repeat until all vertices are marked:

- Select unmarked vertex v with the smallest `distTo[]` value.
- Mark v and relax all edges incident from v.



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

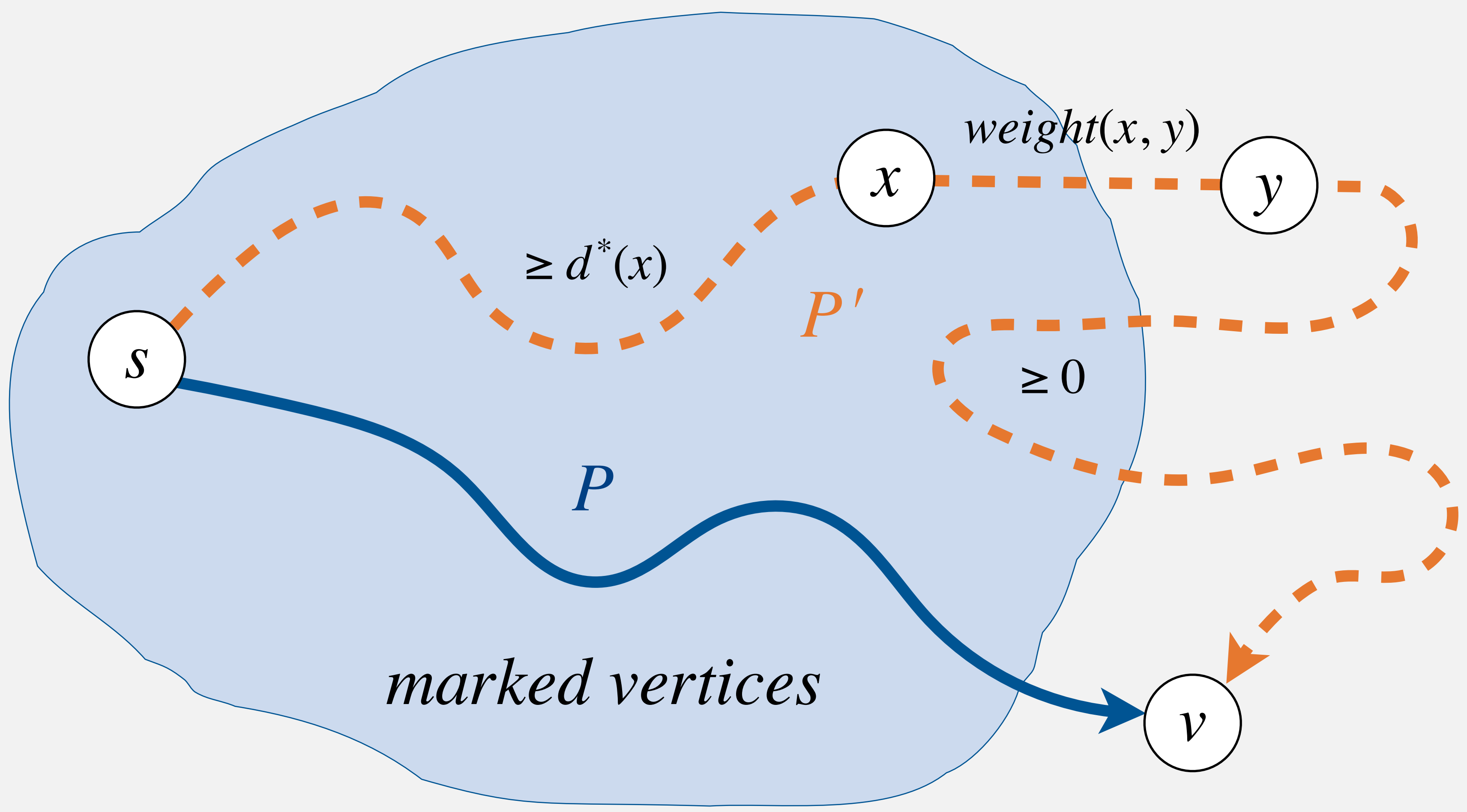
Dijkstra's algorithm: correctness proof

Invariant. For each marked vertex v : `distTo[v]` $= d^*(v)$.

length of shortest path from s to v

Pf. [by induction on number of marked vertices]

- Let v be next vertex marked.
- Let P be the path from s to v of length `distTo[v]`.
- Consider any other path P' from s to v.
- Let $x \rightarrow y$ be first edge in P' with x marked and y unmarked.
- P' is already as long as P by the time it reaches y :



by construction

	length(P) =	`distTo[v]`
Dijkstra chose v instead of y →	≤	`distTo[y]`
relax vertex x →	≤	`distTo[x]` + $weight(x, y)$
induction →	=	$d^*(x)$ + $weight(x, y)$
P' is a path from s to x, followed by edge $x \rightarrow y$, followed by non-negative edges →	≤	length(P') ▪

Dijkstra's algorithm: correctness proof

Invariant. For each marked vertex v : `distTo[v]` $= d^*(v)$.

length of shortest path from s to v

Corollary 1. Dijkstra's algorithm computes shortest path distances.

Corollary 2. Dijkstra's algorithm relaxes vertices in increasing order of distance from s.

generalizes level-order traversal and breadth-first search

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
         int v = pq.delMin();
         for (DirectedEdge e : G.adj(v))
            relax(e);
      }
   }
}
```

PQ that supports decreasing the key (stay tuned)

PQ contains the unmarked vertices with finite distTo[] values

relax vertices in order of distance from s

Dijkstra's algorithm: Java implementation

When relaxing an edge, also update PQ:

- Found first path from s to w: add w to PQ.
- Found better path from s to w: decrease key of w in PQ.

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
       if (!pq.contains(w)) pq.insert(w, distTo[w]);
       else                 pq.decreaseKey(w, distTo[w]);
   }
}
```

← update PQ

Q. How to implement DECREASE-KEY operation in a priority queue?

Indexed priority queue (Section 2.4)

Associate an index between 0 and $n - 1$ with each key in a priority queue.

- Insert a key associated with a given index.
- Delete a minimum key and return associated index.
- Decrease the key associated with a given index.

for Dijkstra's algorithm:
$n = V$,
index = vertex,
key = distance from s

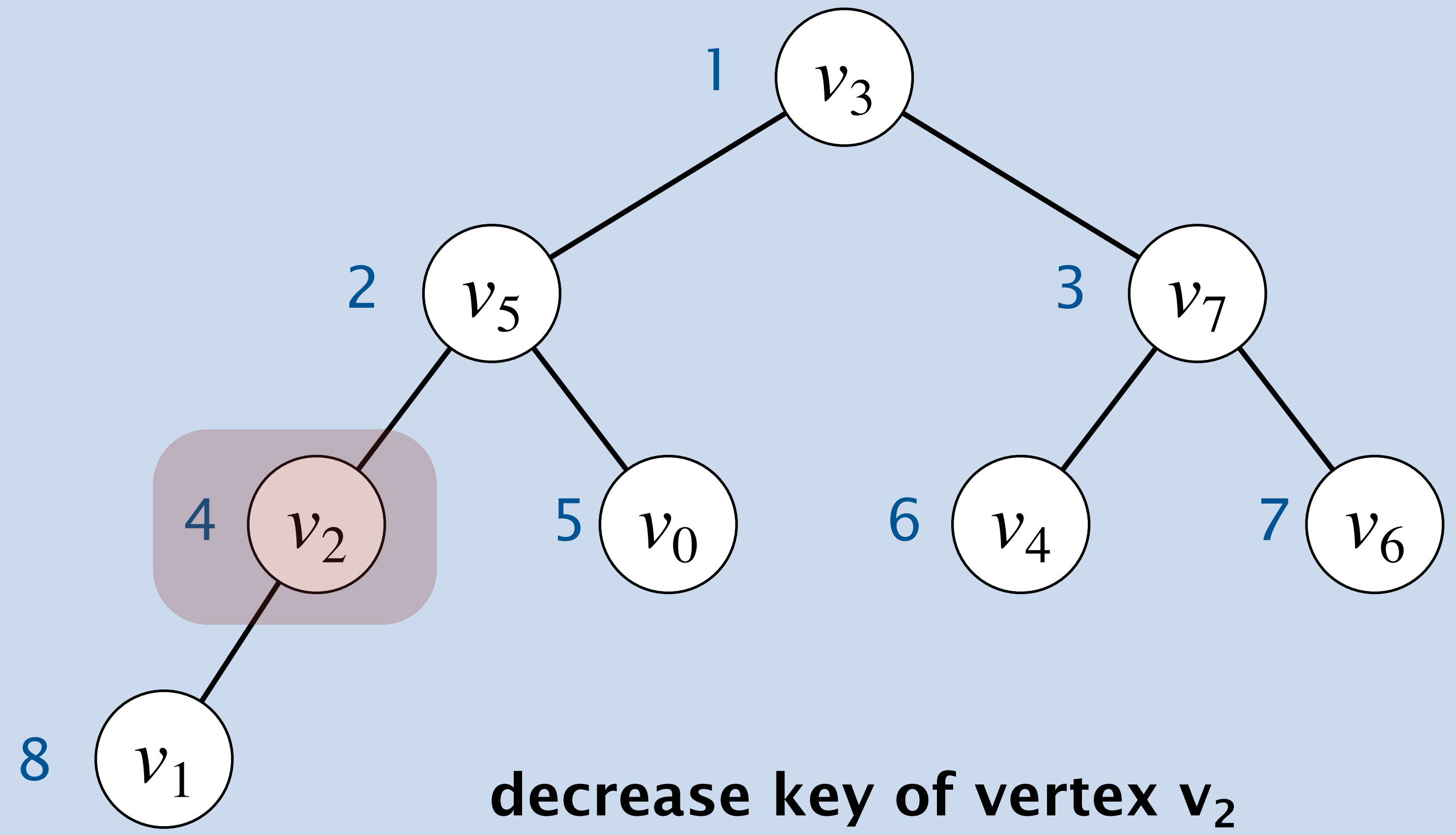
public class IndexMinPQ<Key extends Comparable<Key>>

	IndexMinPQ(int n)	*create PQ with indices* $0, 1, \ldots, n - 1$
void	insert(int i, Key key)	*associate key with index i*
int	delMin()	*remove min key and return associated index*
void	decreaseKey(int i, Key key)	*decrease the key associated with index i*
boolean	isEmpty()	*is the priority queue empty?*
	⋮	⋮

DECREASE-KEY IN A BINARY HEAP

Goal. Implement DECREASE-KEY operation in a binary heap.

	0	1	2	3	4	5	6	7	8
pq[]	–	v_3	v_5	v_7	v_2	v_0	v_4	v_6	v_1



decrease key of vertex v_2

Decrease-Key in a Binary Heap

Goal. Implement Decrease-Key operation in a binary heap.

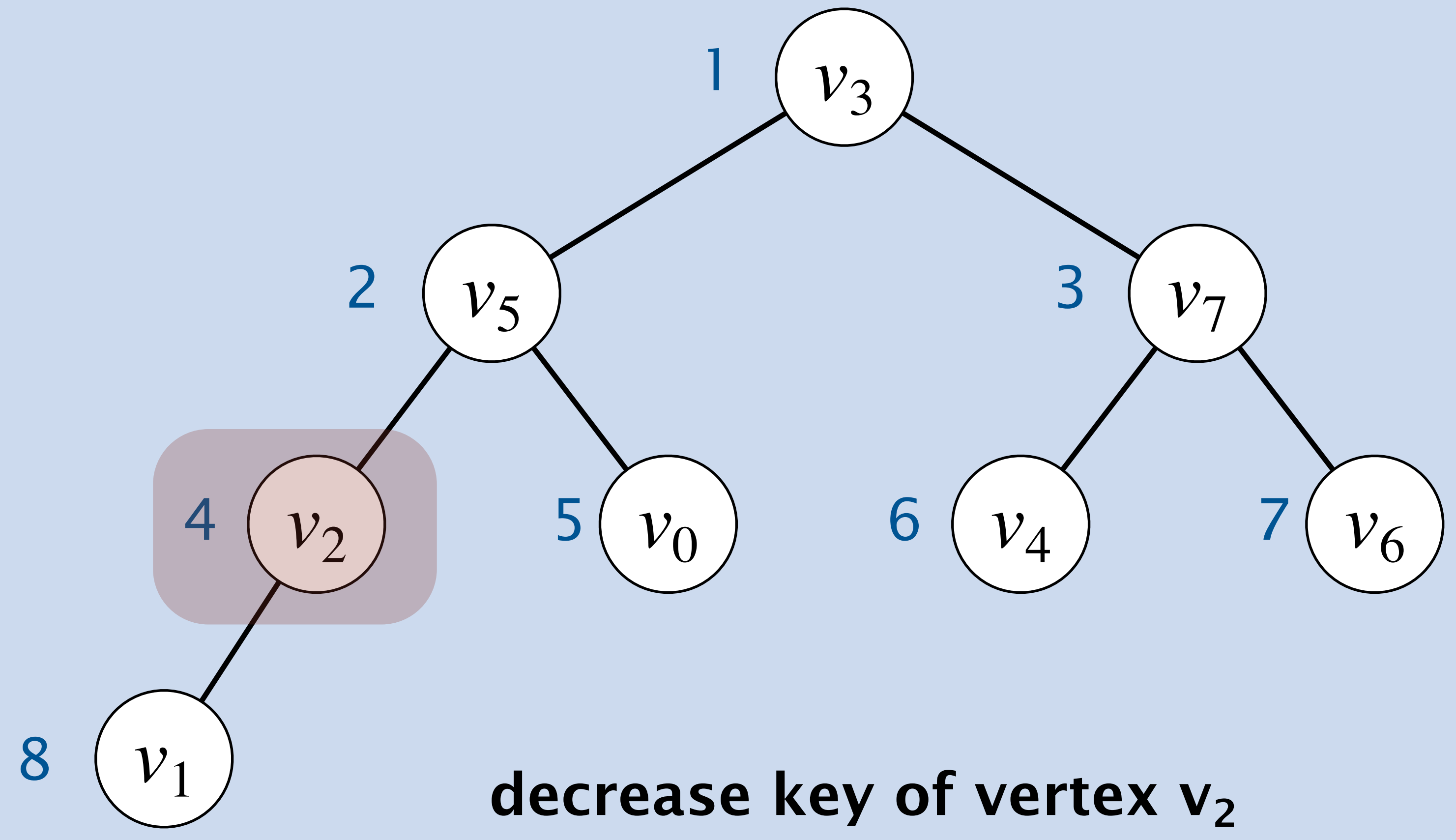
Solution.

- Find vertex in heap. How?
- Change priority of vertex and call `swim()` to restore heap invariant.

Extra data structure. Maintain an inverse array `qp[]` that maps from the vertex to the binary heap node index.

	0	1	2	3	4	5	6	7	8
pq[]	–	v_3	v_5	v_7	v_2	v_0	v_4	v_6	v_1
qp[]	5	8	4	1	6	2	4	3	–
keys[]	1.0	2.0	3.0	0.0	6.0	8.0	4.0	2.0	–

vertex 2 has priority 3.0
and is at heap index 4



decrease key of vertex v_2

Dijkstra's algorithm: which priority queue?

Number of PQ operations: V INSERT, V DELETE-MIN, $\leq E$ DECREASE-KEY.

PQ implementation	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

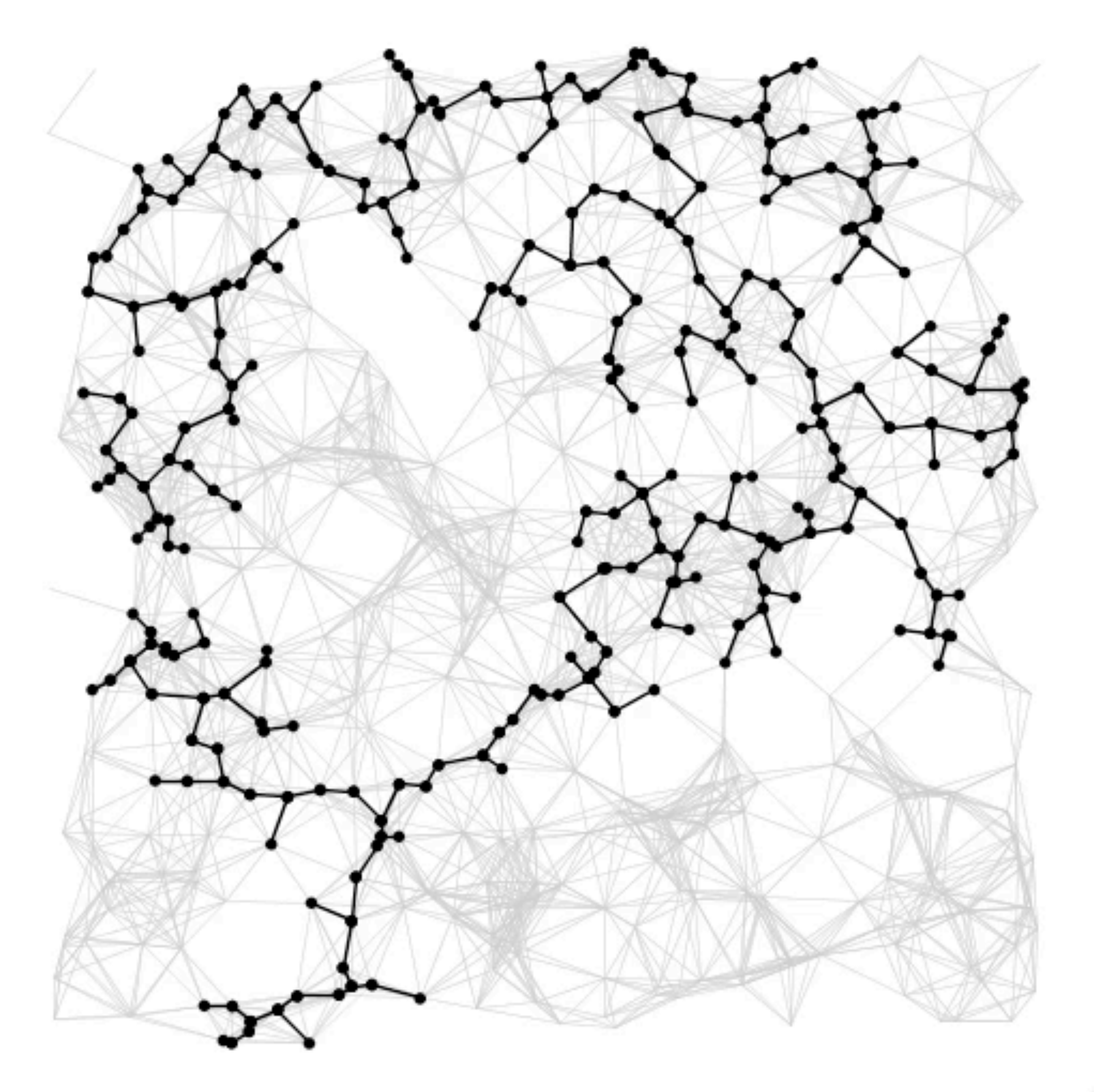
Bottom line.

- Array implementation optimal for complete digraphs.
- Binary heap much faster for sparse digraphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

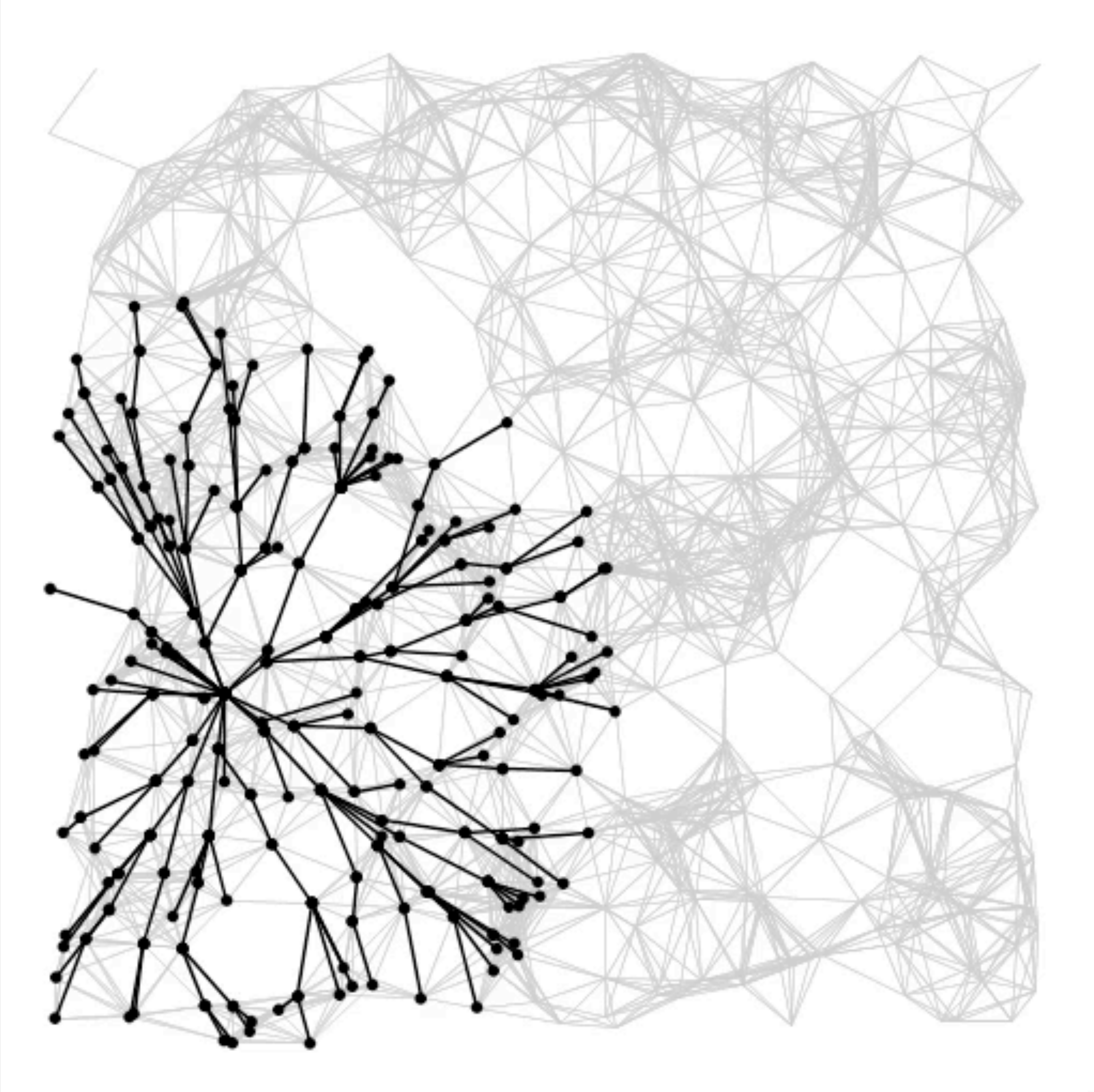
Priority-first search

Observation. Prim and Dijkstra are essentially the same algorithm.

- Prim: Choose next vertex that is closest to any vertex in the tree (via an undirected edge).
- Dijkstra: Choose next vertex that is closest to the source vertex (via a directed path).



Prim's algorithm



Dijkstra's algorithm

Algorithms for shortest paths

Variations on a theme: vertex relaxations.

- Bellman–Ford: relax all vertices; repeat $V - 1$ times.
- Dijkstra: relax vertices in order of distance from s.
- Topological sort: relax vertices in topological order. ← see Section 4.4 and next lecture

algorithm	worst-case running time	negative weights †	directed cycles
Bellman-Ford	$E\ V$	✔	✔
Dijkstra	$E \log V$		✔
topological sort	E	✔	

† no negative cycles

Which shortest paths algorithm to use?

Select algorithm based on properties of edge-weighted digraph.

- Negative weights (but no "negative cycles"): Bellman–Ford.
- Non-negative weights: Dijkstra.
- DAG: topological sort.

algorithm	worst-case running time	negative weights †	directed cycles
Bellman–Ford	$E\ V$	✔	✔
Dijkstra	$E \log V$		✔
topological sort	E	✔	

† no negative cycles