https://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

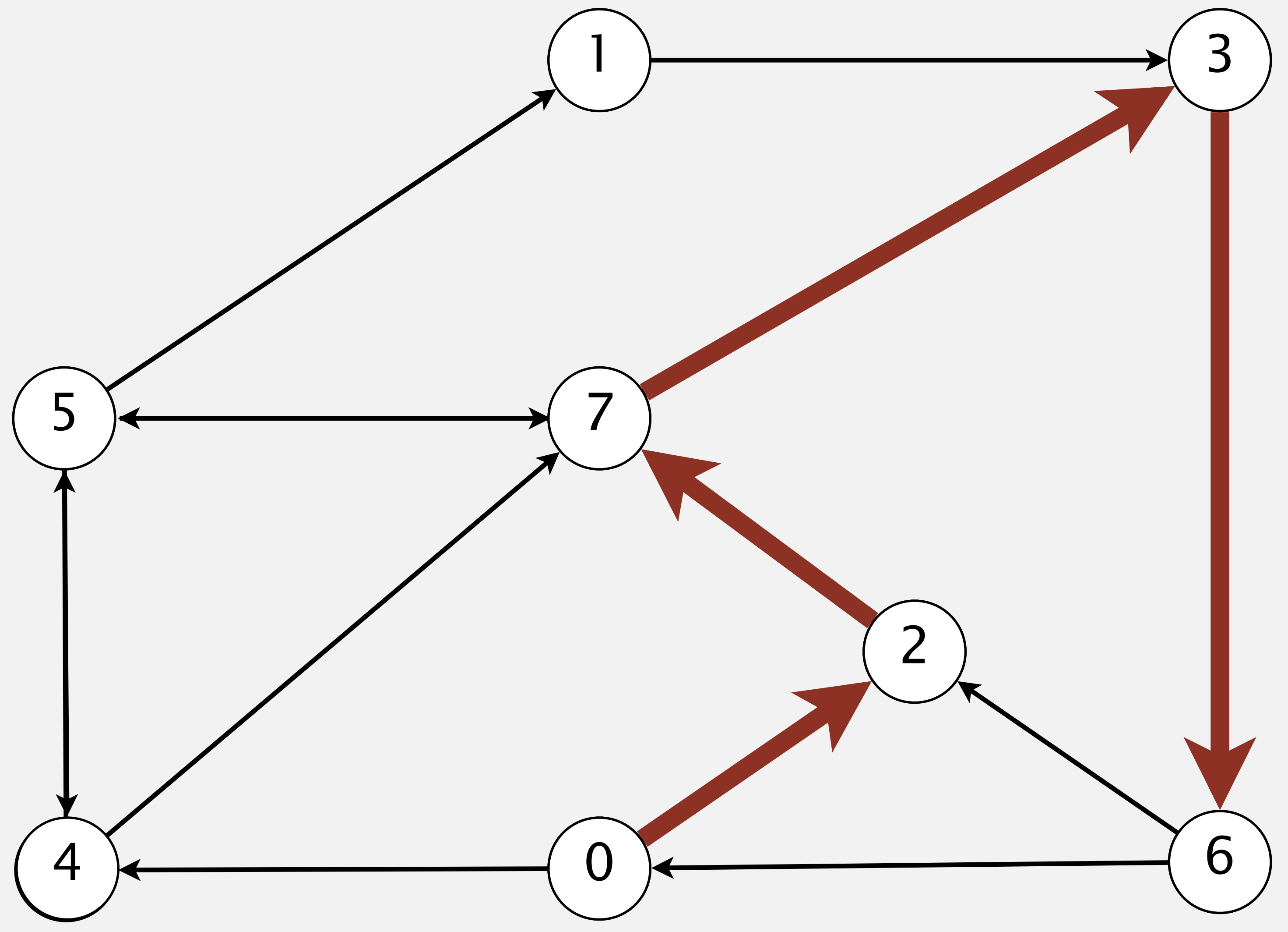
- *properties*
- *APIs*
- *Bellman–Ford algorithm*
- *Dijkstra's algorithm*

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
```



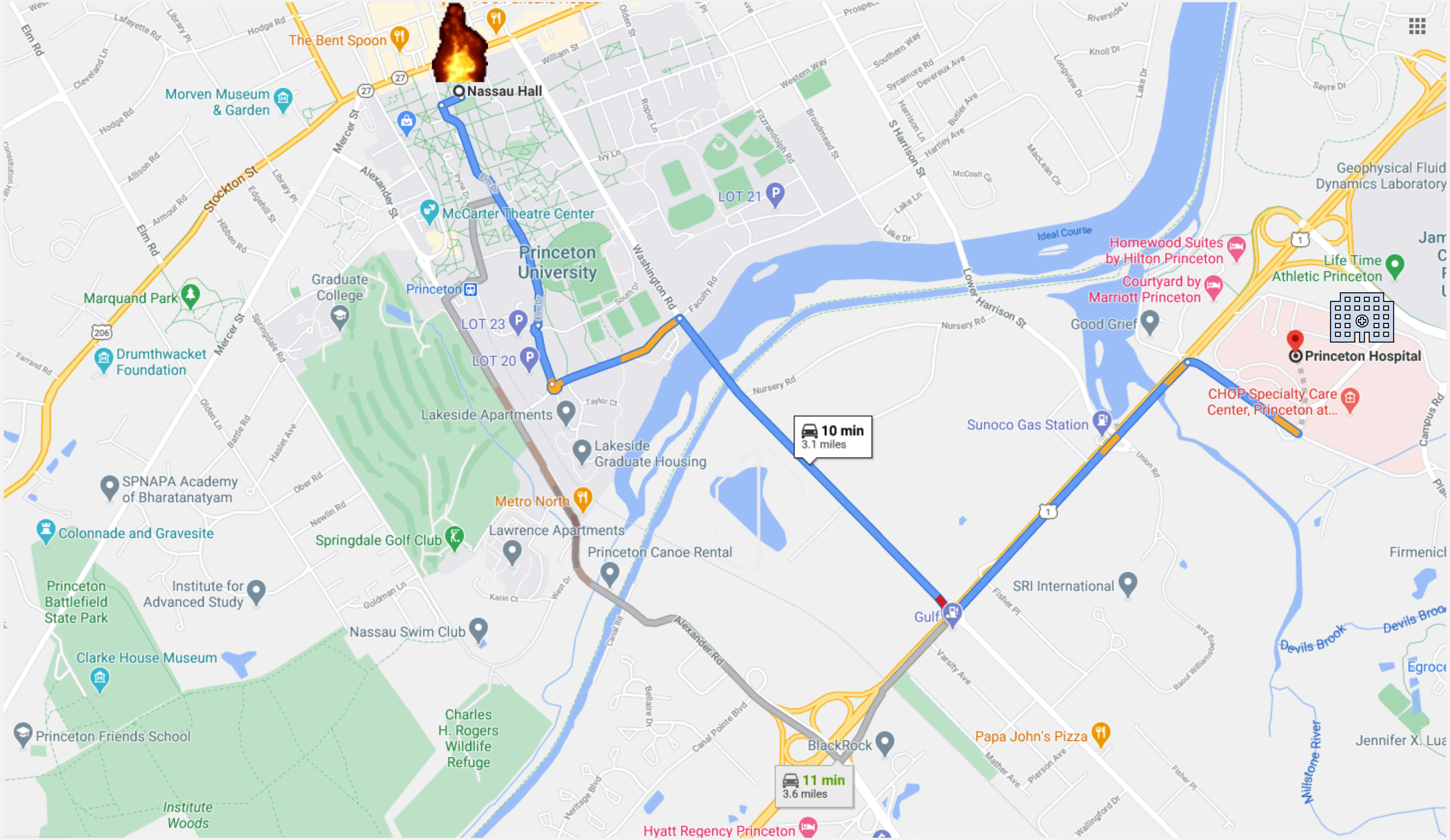
shortest path from 0 to 6

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

length of path = 1.51

$(0.26 + 0.34 + 0.39 + 0.52\ )$

Google maps



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

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

Restrictions on edge weights?

- Non-negative weights.
- Euclidean weights.
- Arbitrary weights.

we assume this in today's lecture
(except as noted)

Cycles?

- No directed cycles.
- No "negative cycles."

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

Simplifying assumption. Each vertex is reachable from s.

Which variant in car GPS?

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

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

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

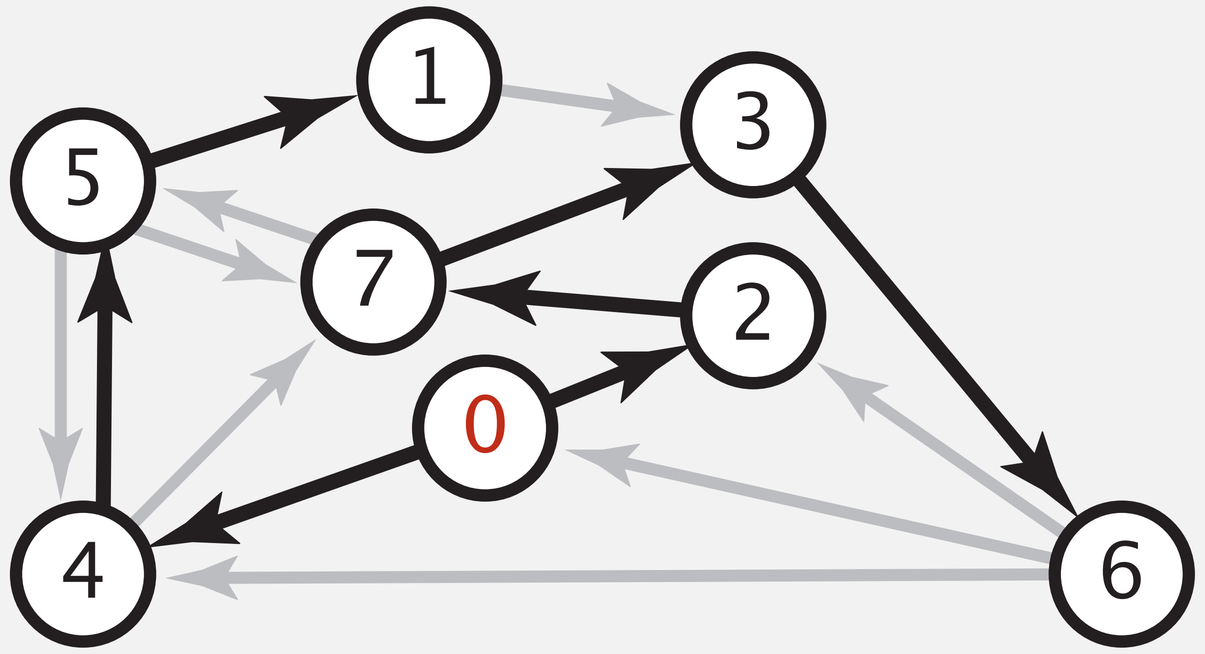
Goal. Find a shortest path from s to every other vertex.

Observation 1. There exists a shortest path from s to v that is simple. ← no repeated vertices $\Rightarrow \le V - 1$ edges

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

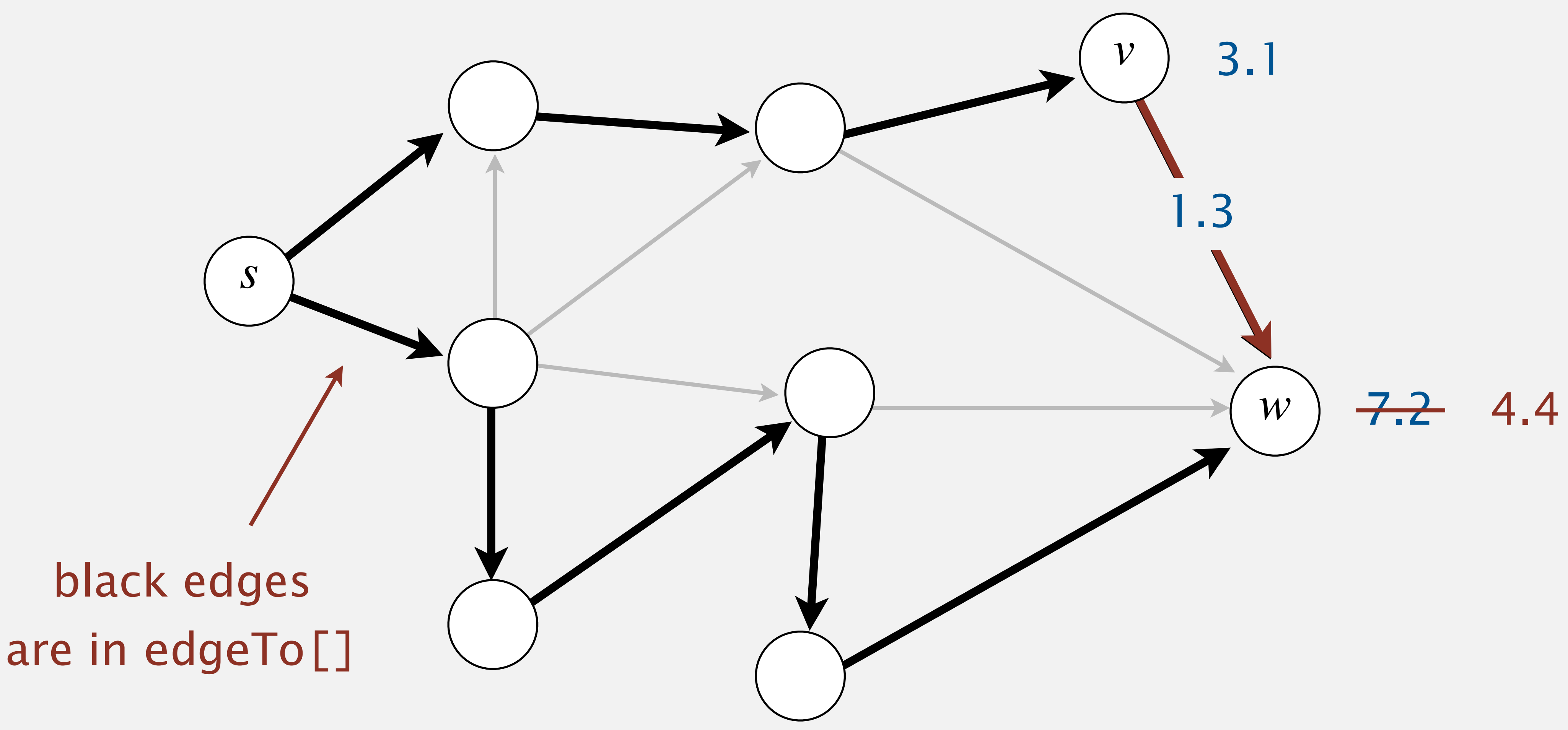
parent-link representation

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `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 $e = v \rightarrow w$ yields shorter path from s to w, via v, update `distTo[w]` and `edgeTo[w]`.

relax edge e = v→w



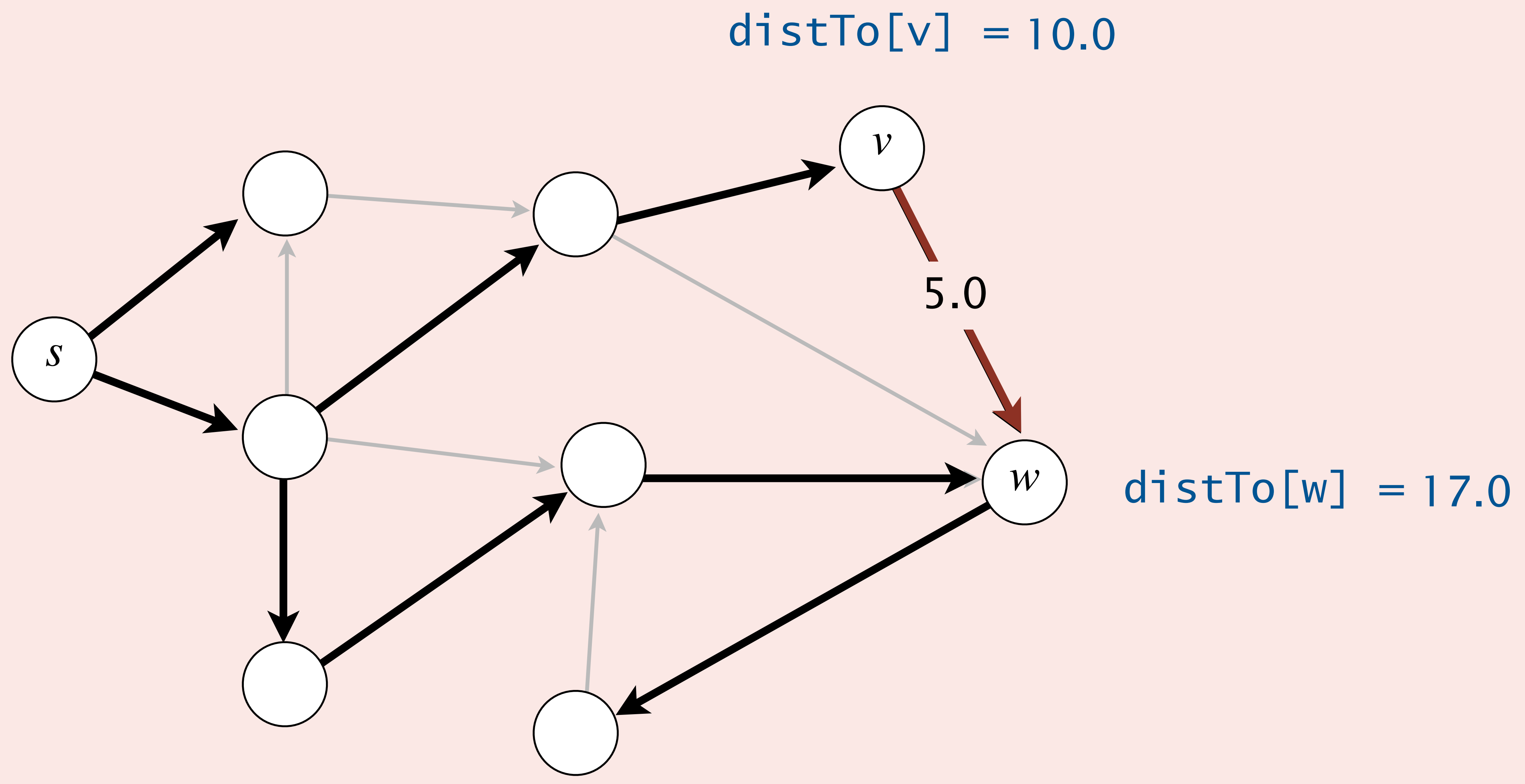
What are the values of `distTo[v]` and `distTo[w]` after relaxing $e = v \to w$?

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 done:

- **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 done:

- **Relax any edge.**

Efficient implementations.

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

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 $e = v \rightarrow w$.

```
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*
	⋮	⋮

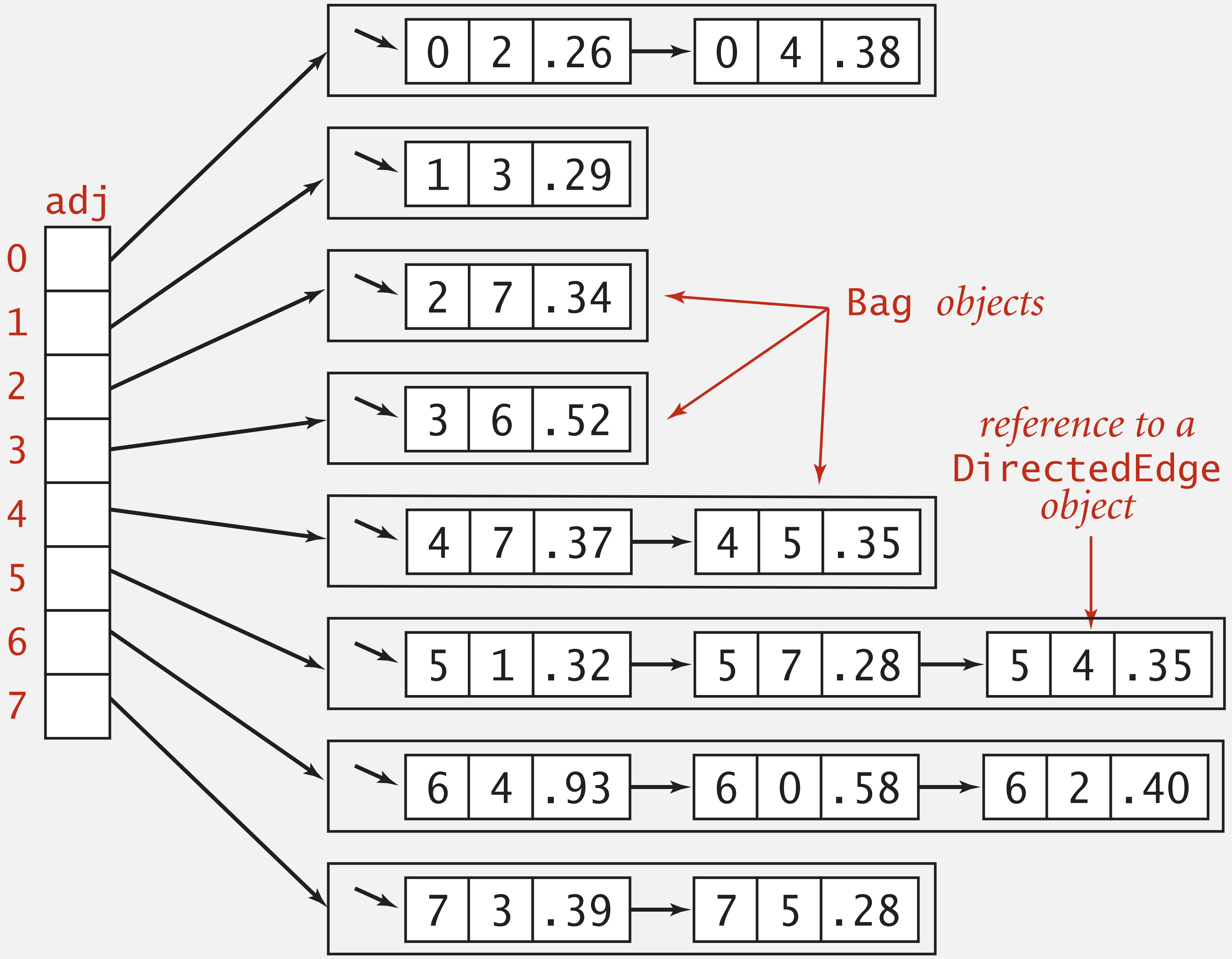
Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V → 8

15 ← *E*

```
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(), w = e.to();
     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.**

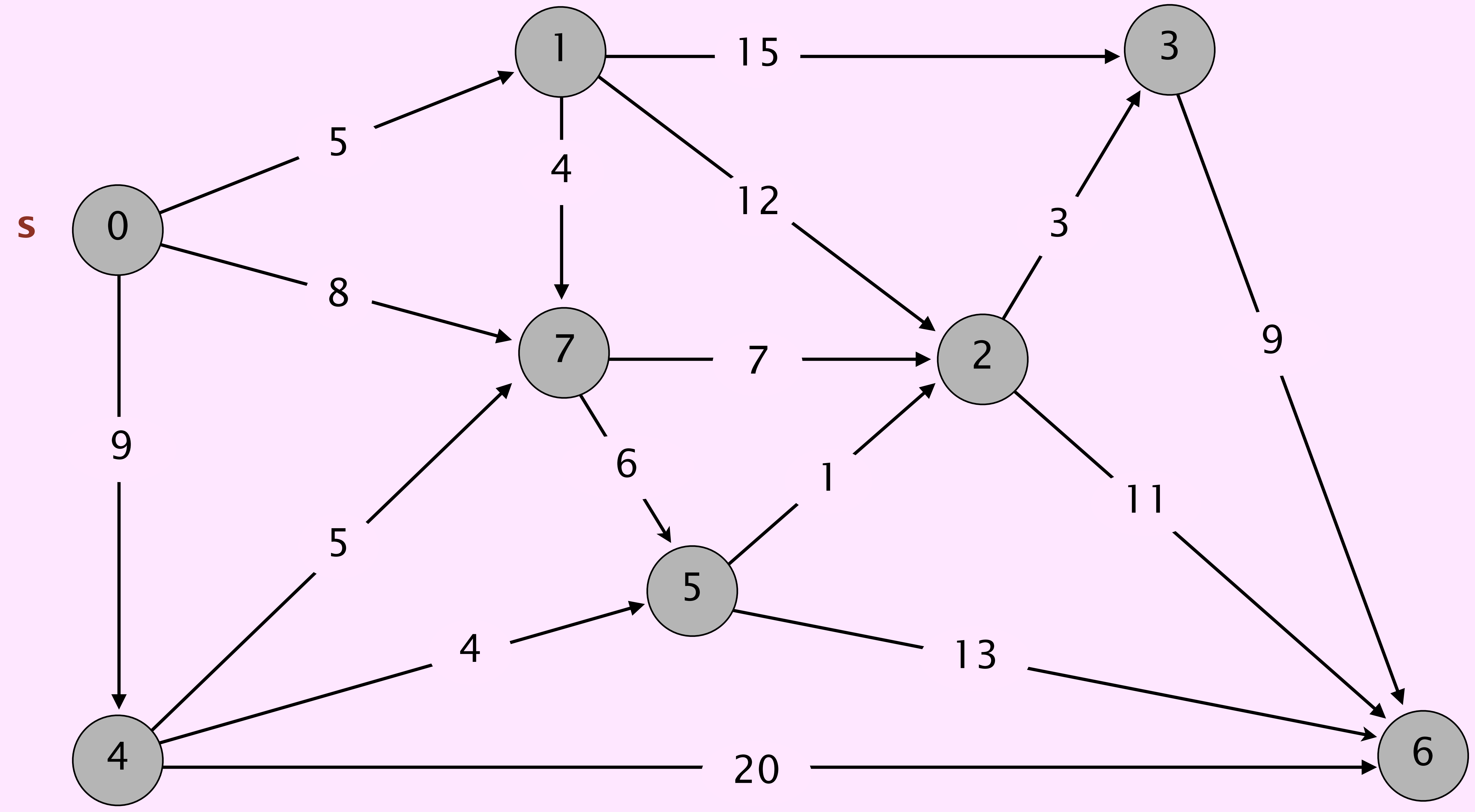
```
for (int i = 1; i < G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

pass i (relax each edge)

Running time. Algorithm takes $\Theta(E\ V)$ time in both best- and worst-case.

Bellman–Ford algorithm demo

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

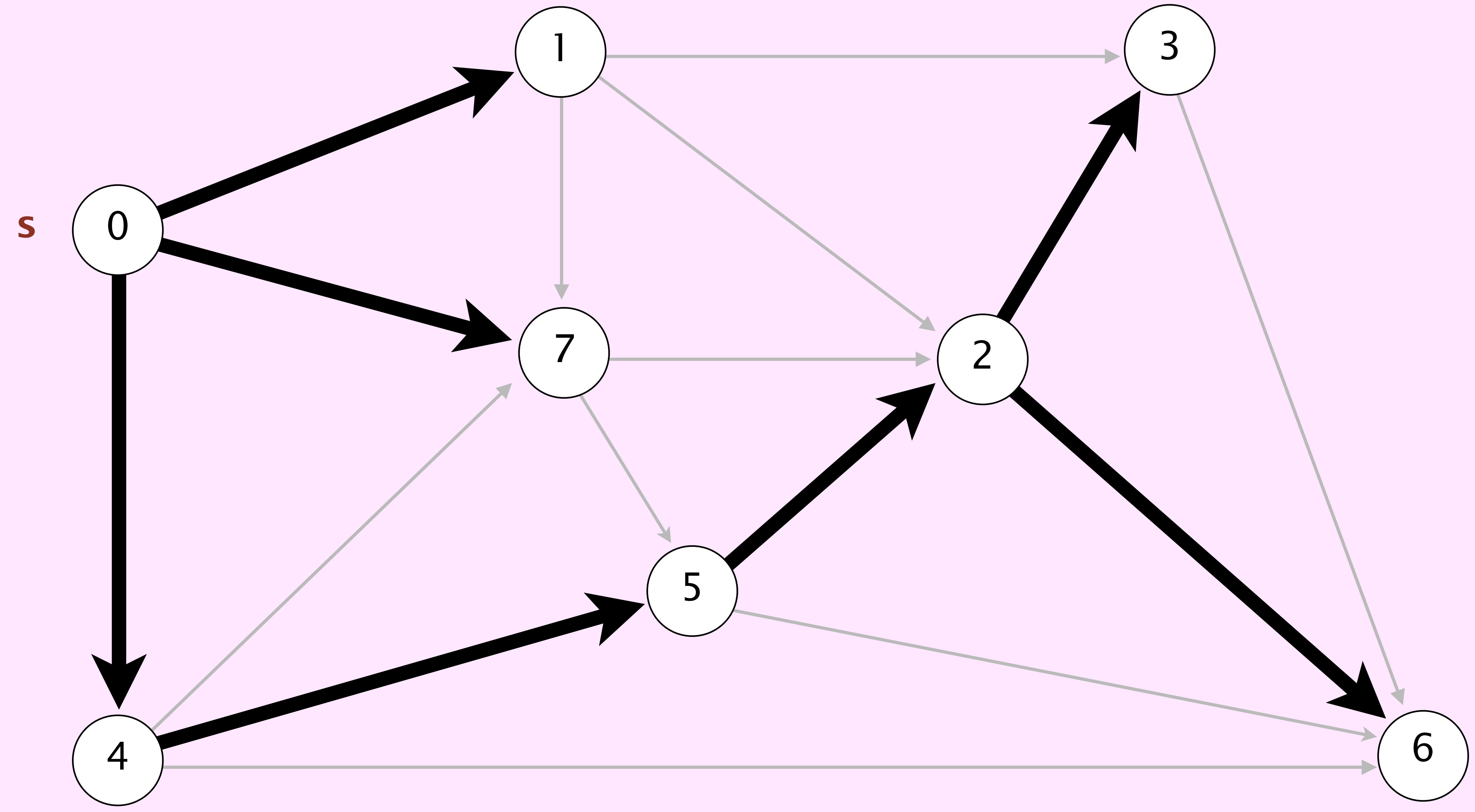


an edge-weighted digraph

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

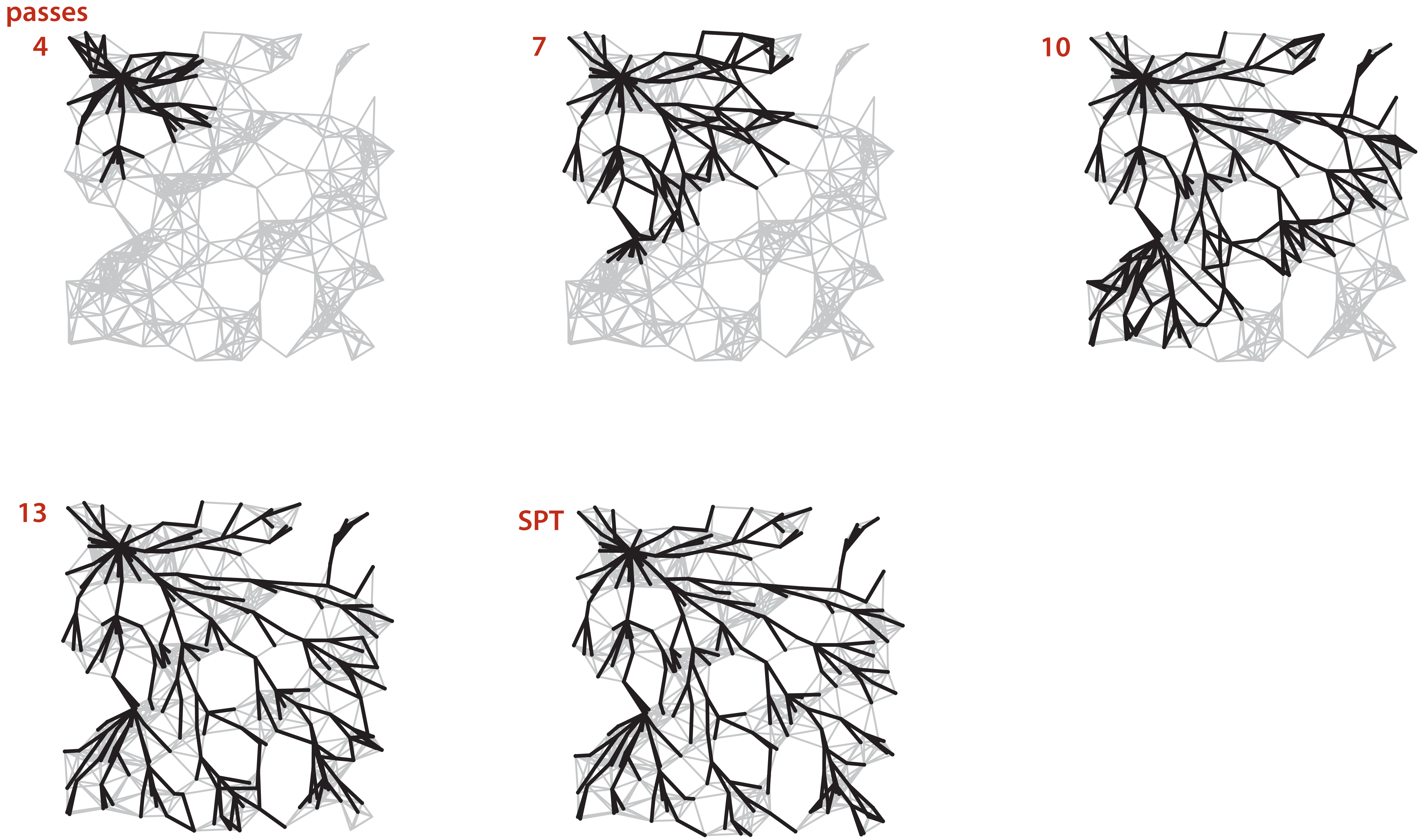
Repeat $V-1$ times: relax all E edges.



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

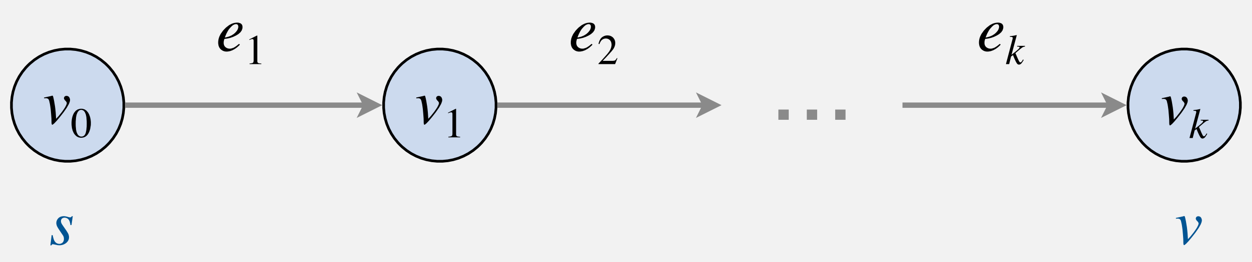
shortest-paths tree from vertex s

Bellman–Ford algorithm: visualization



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



Pf. [by induction on number of passes i]

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

$$\texttt{distTo[}v_{i+1}\texttt{]} \le \texttt{distTo[}v_i\texttt{]} + weight(e_{i+1}) \quad \longleftarrow \text{edge relaxation}$$

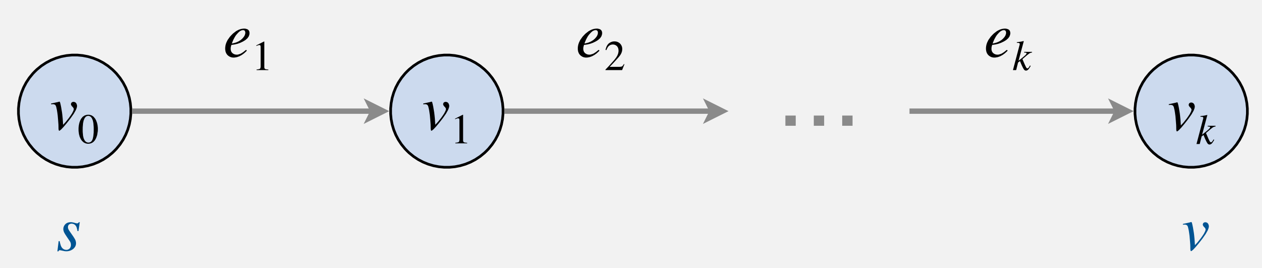
$$\le weight(e_1) + weight(e_2) + \cdots + weight(e_i) + weight(e_{i+1}). \quad \longleftarrow \text{inductive hypothesis}$$

- 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.

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.

- There exists a shortest path P^* from s to v with at most $V - 1$ 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`]` $= length(P^*)$. ▪

Suppose that `distTo[v]` does not change during pass i of Bellman–Ford.

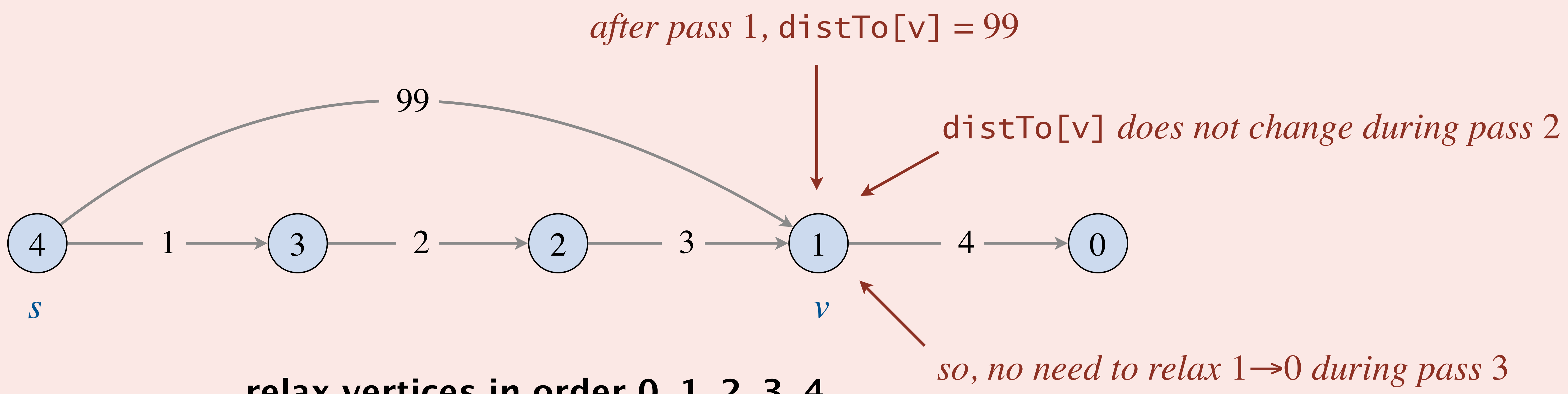
Which of the following are true?

A. `distTo[v]` is the length of a shortest path from s to v.

B. Not necessary to relax any edges incident to v in pass $i + 1$.

C. Not necessary to relax any edges incident from v in pass $i + 1$.

D. All of the above.



relax vertices in order 0, 1, 2, 3, 4

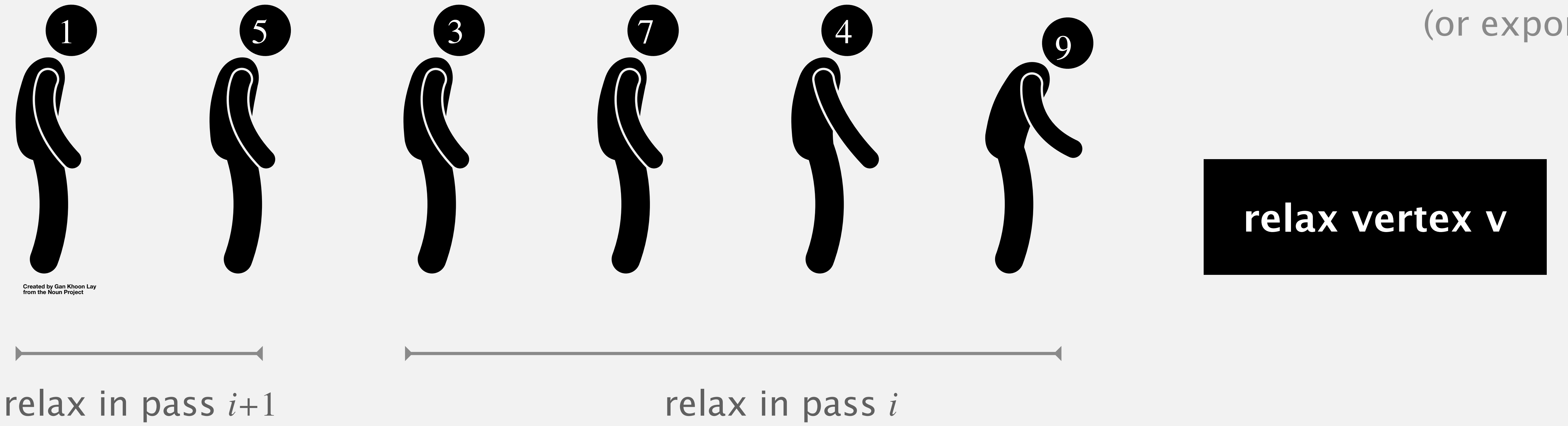
Bellman–Ford algorithm: practical improvement

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!)



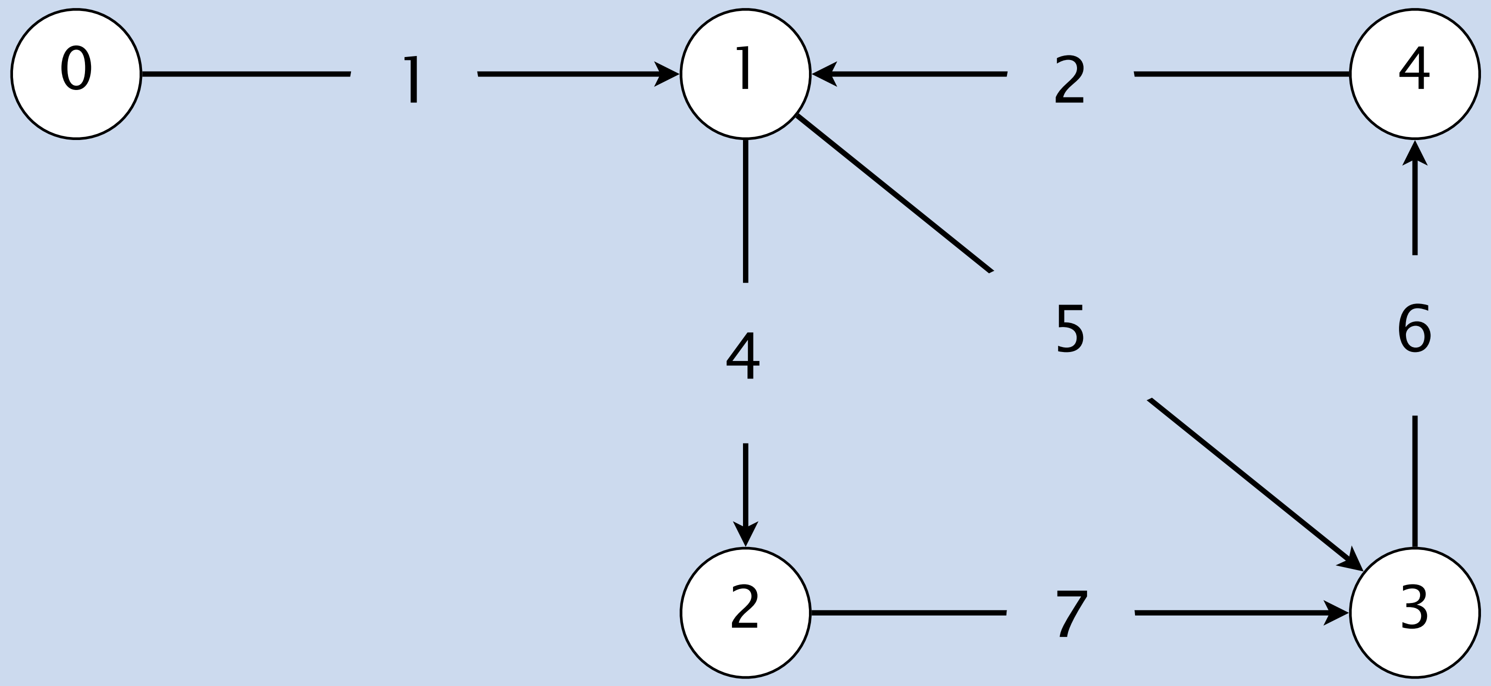
Impact.

- 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.

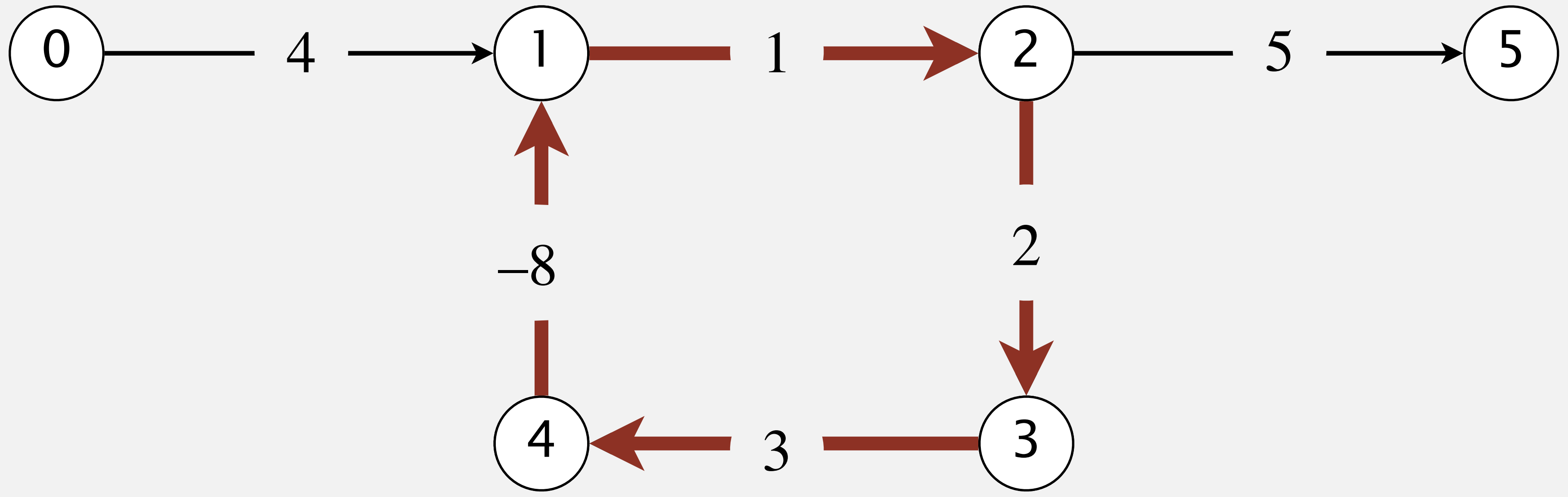


longest simple path from 0 to 4: 0→1→2→3→4

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

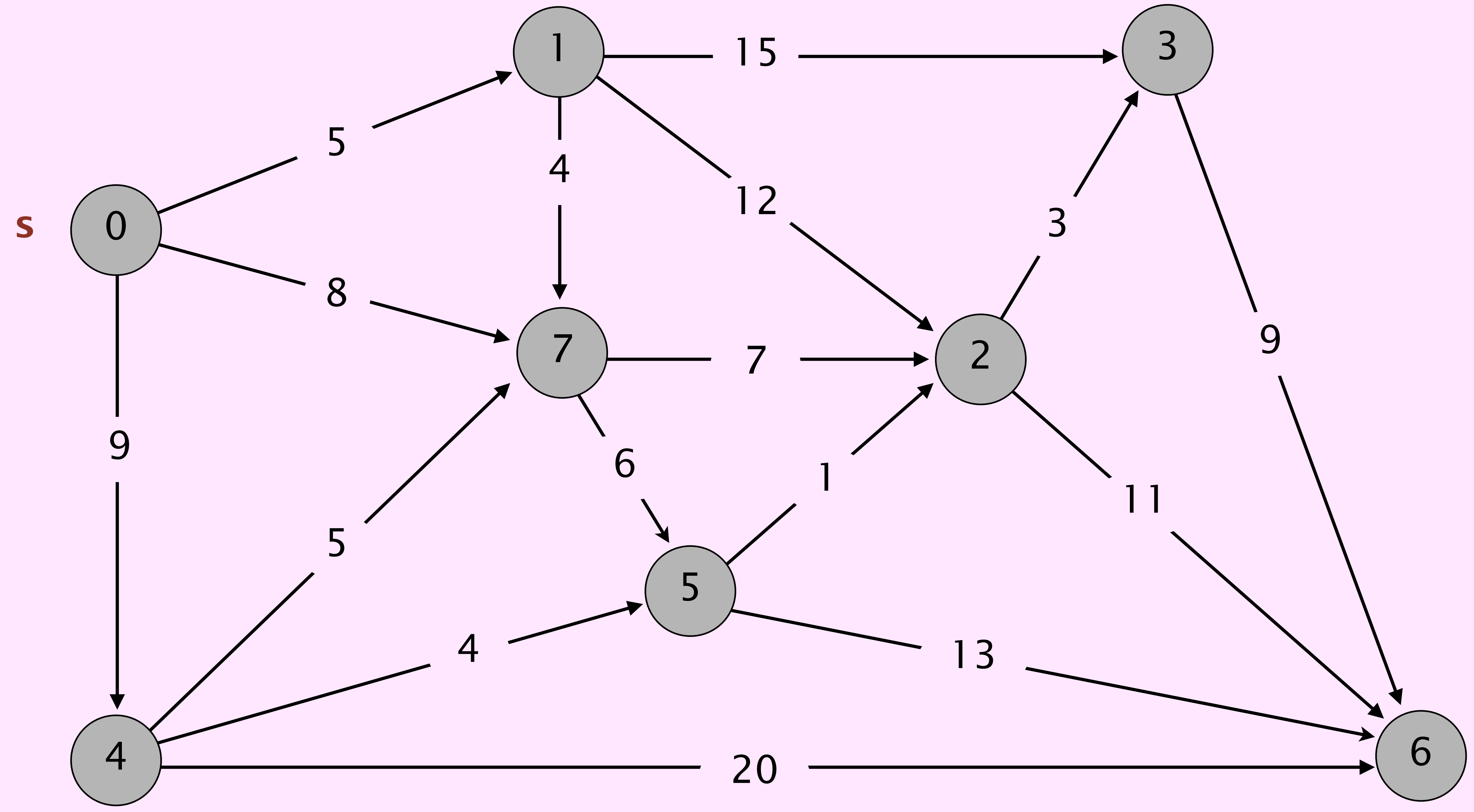
4.4 Shortest Paths

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

Edsger W. Dijkstra: select quotes

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.

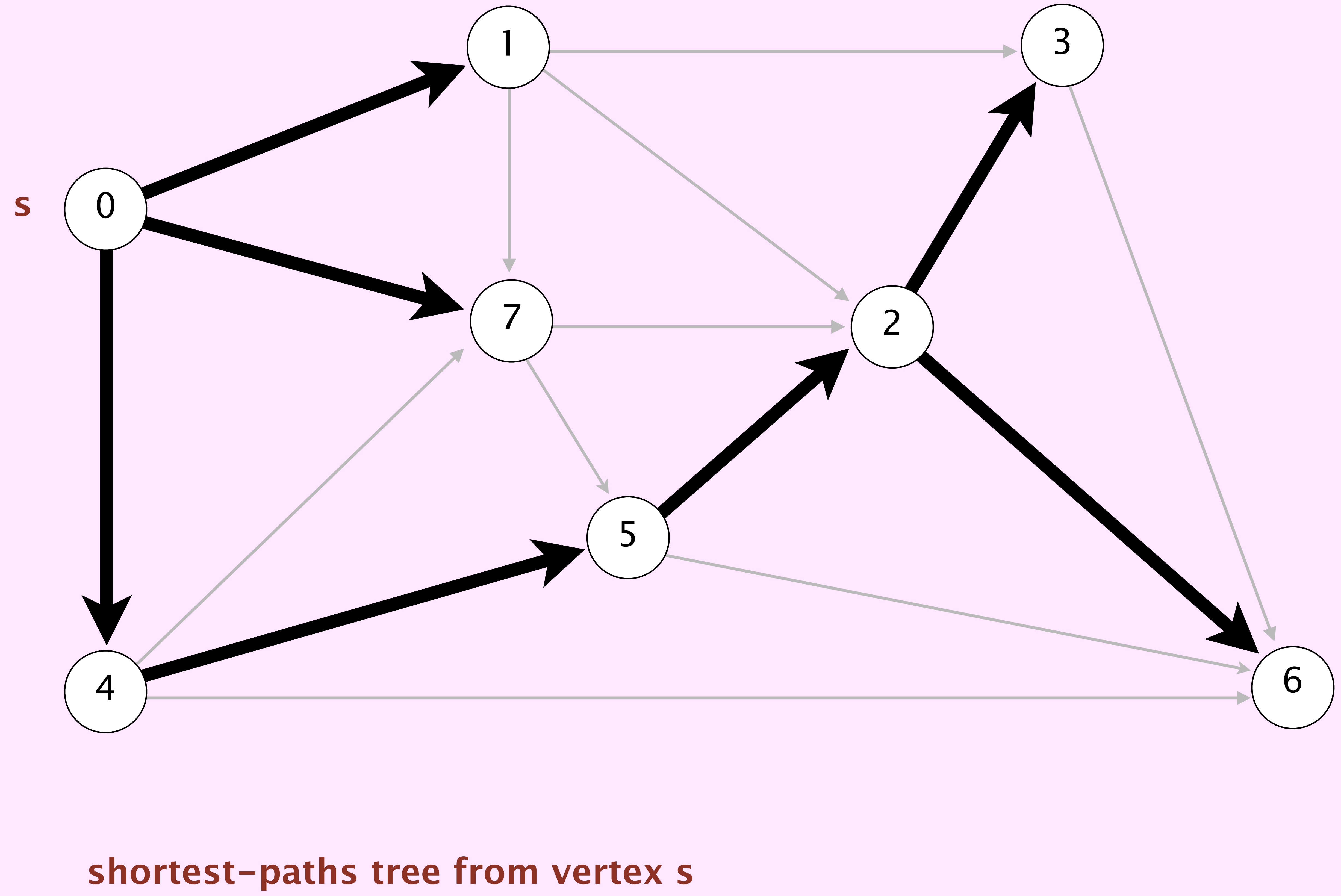


an edge-weighted digraph

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

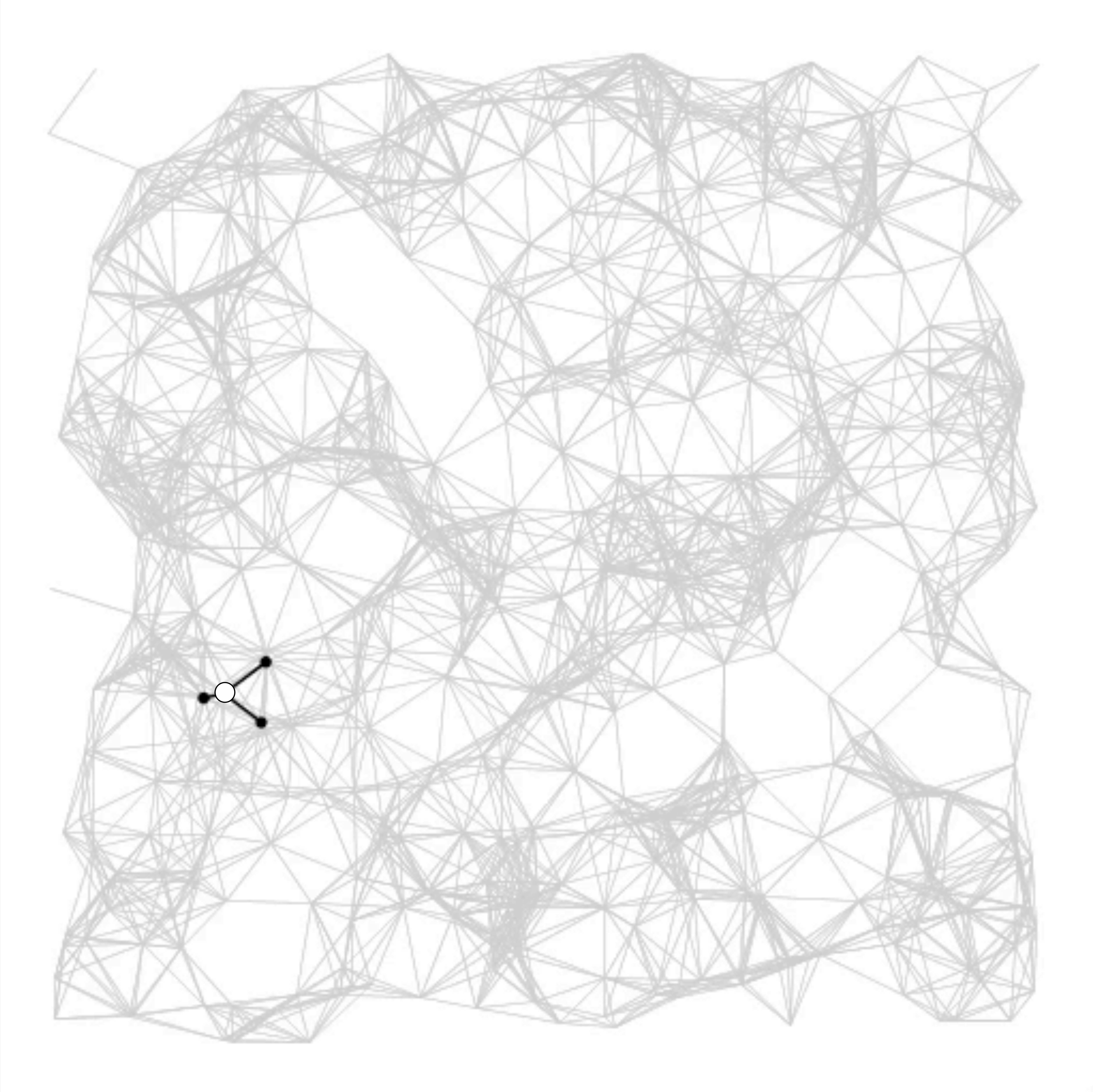
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.

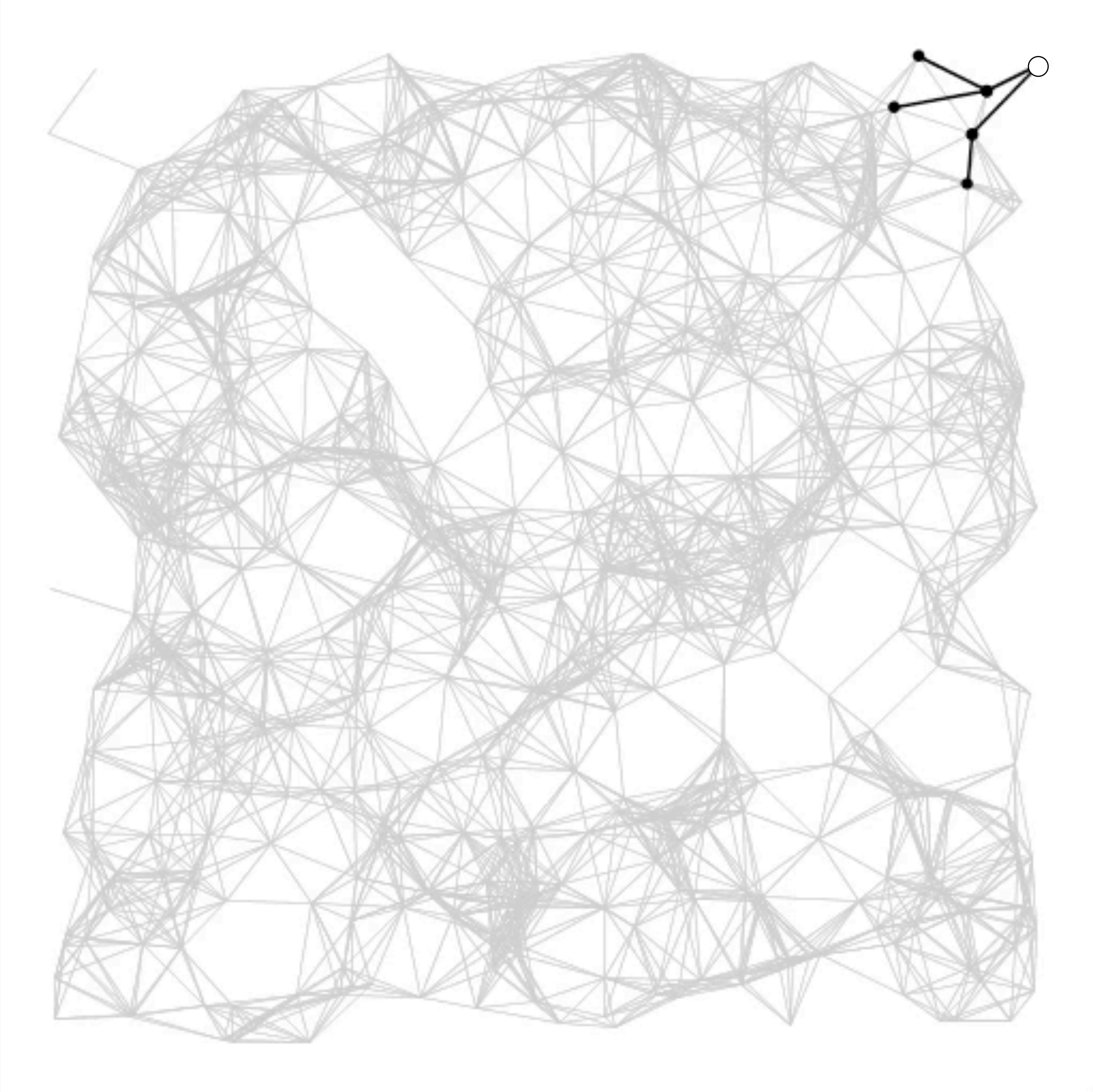


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm visualization



Dijkstra's algorithm visualization



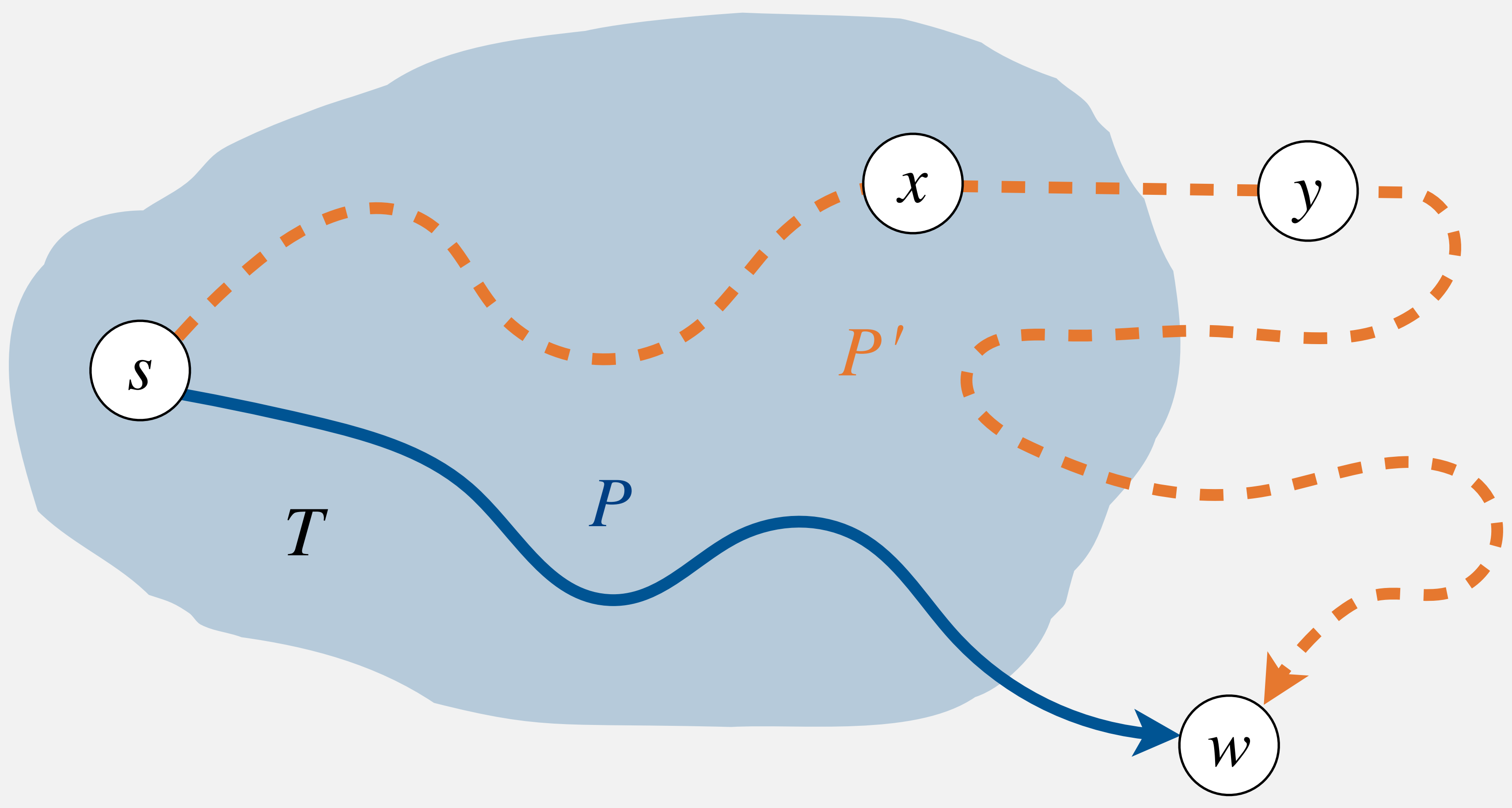
Dijkstra's algorithm: correctness proof

Invariant. For each vertex v in T, `distTo[v]` $= d^*(v)$.

length of shortest path from s to v

Pf. [by induction on $|T|$]

- Let w be next vertex added to T.
- Let P be the path from s to w of length `distTo[w]`.
- Consider any other path P' from s to w.
- Let $x \rightarrow y$ be first edge in P' that leaves T.
- P' is no shorter than P:



by construction

	length(P)	= `distTo[w]`
Dijkstra chose w instead of y ⟶		≤ `distTo[y]`
relax vertex x ⟶		≤ `distTo[x]` + $weight(x, y)$
induction ⟶		= $d^*(x)$ + $weight(x, y)$
weights are non-negative ⟶		≤ length(P') ▪

Dijkstra's algorithm: correctness proof

Invariant. For each vertex v in T, `distTo[v]` $= d^*(v)$.

length of shortest path from s to v

Corollary. Dijkstra's algorithm computes shortest path distances.

Pf. Upon termination, T contains all vertices (reachable from s).

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)

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

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

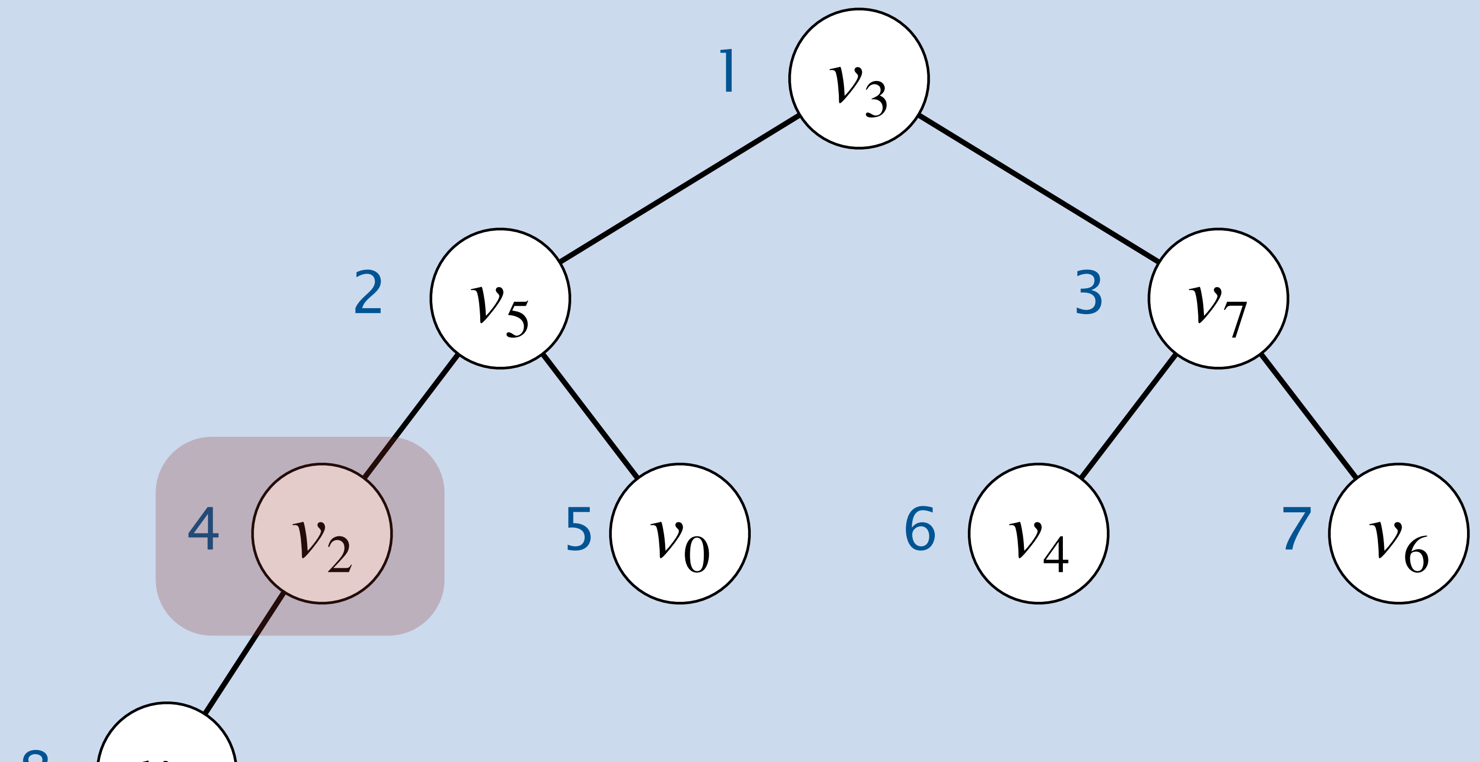
```
public class IndexMinPQ<Key extends Comparable<Key>>
```

	`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.

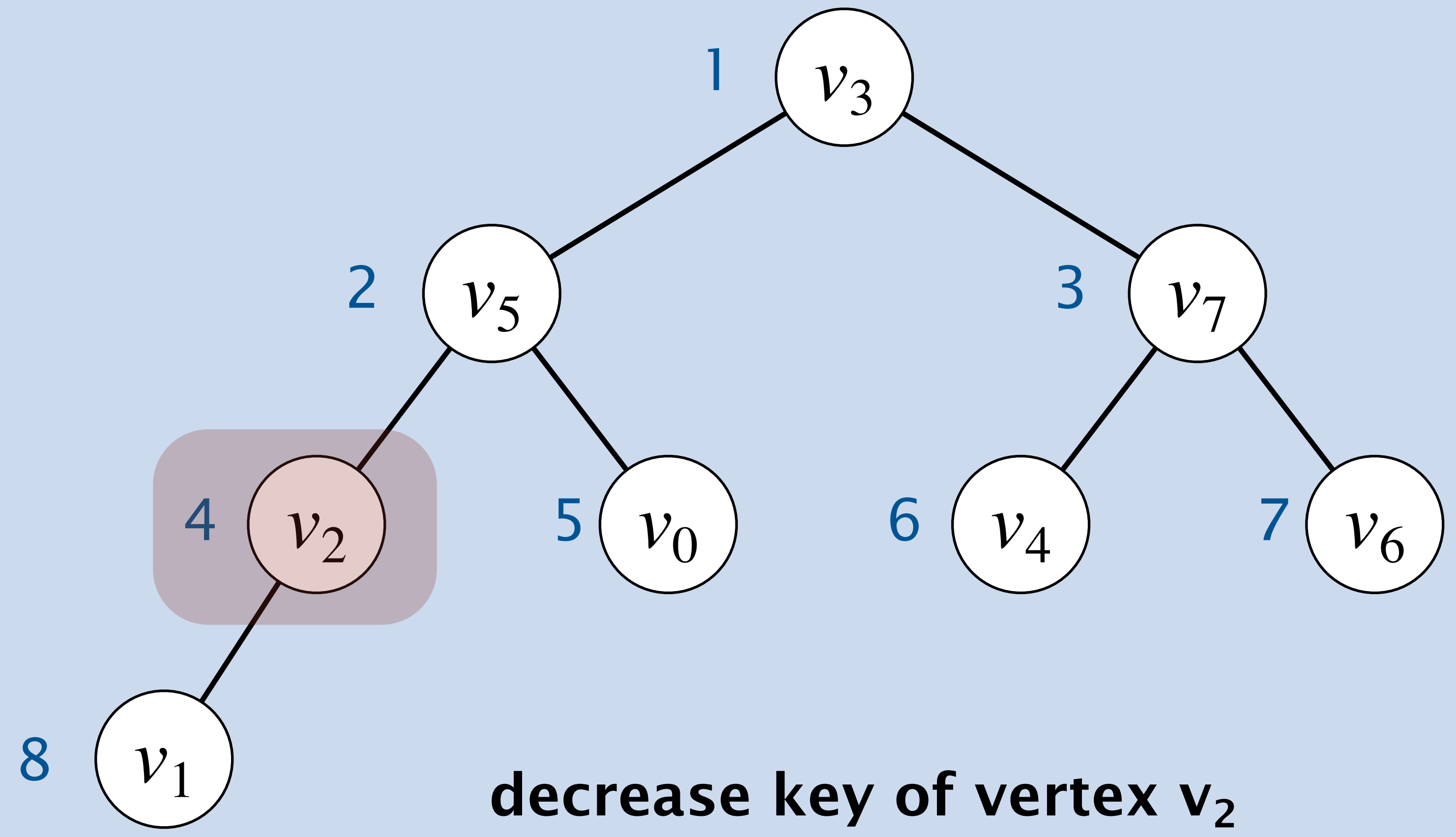
Solution.

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

Extra data structure. Maintain an 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	$1^{\dagger}$	$\log V^{\dagger}$	$1^{\dagger}$	$E + V \log V$

† 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.

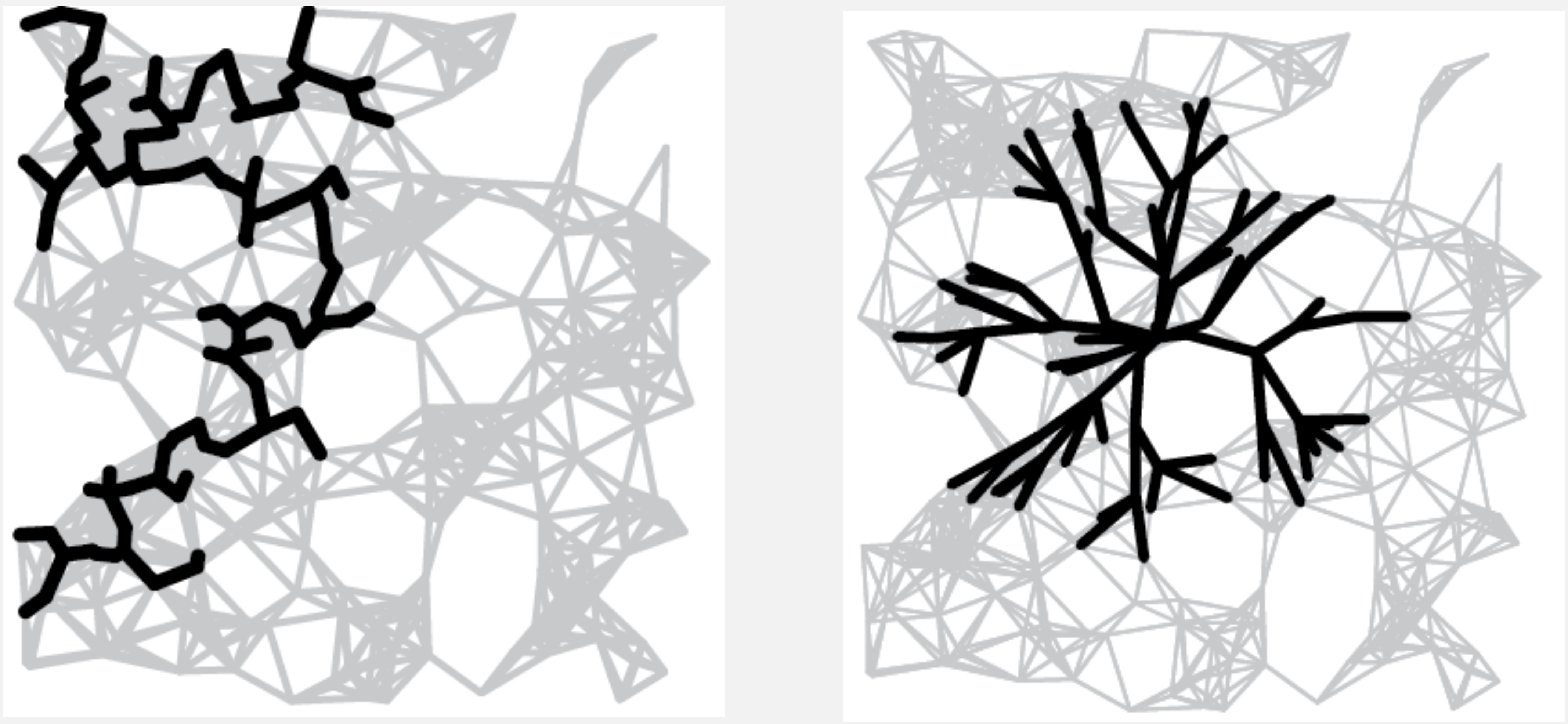
Priority-first search

Dijkstra’s algorithm seems familiar?

- Prim’s algorithm is essentially the same algorithm.
- Both in same family of algorithms.

Main distinction: rule used to choose next vertex for the tree.

- Prim: Closest vertex to the tree (via an undirected edge).
- Dijkstra: Closest vertex to the source (via a directed path).



Note: DFS and BFS are also in same family.

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

https://algs4.cs.princeton.edu

4.4 Shortest Paths

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

Content-aware resizing

Seam carving. [Avidan–Shamir] Resize an image without distortion for display on cell phones and web browsers.

https://www.youtube.com/watch?v=vIFCV2spKtg

Content-aware resizing

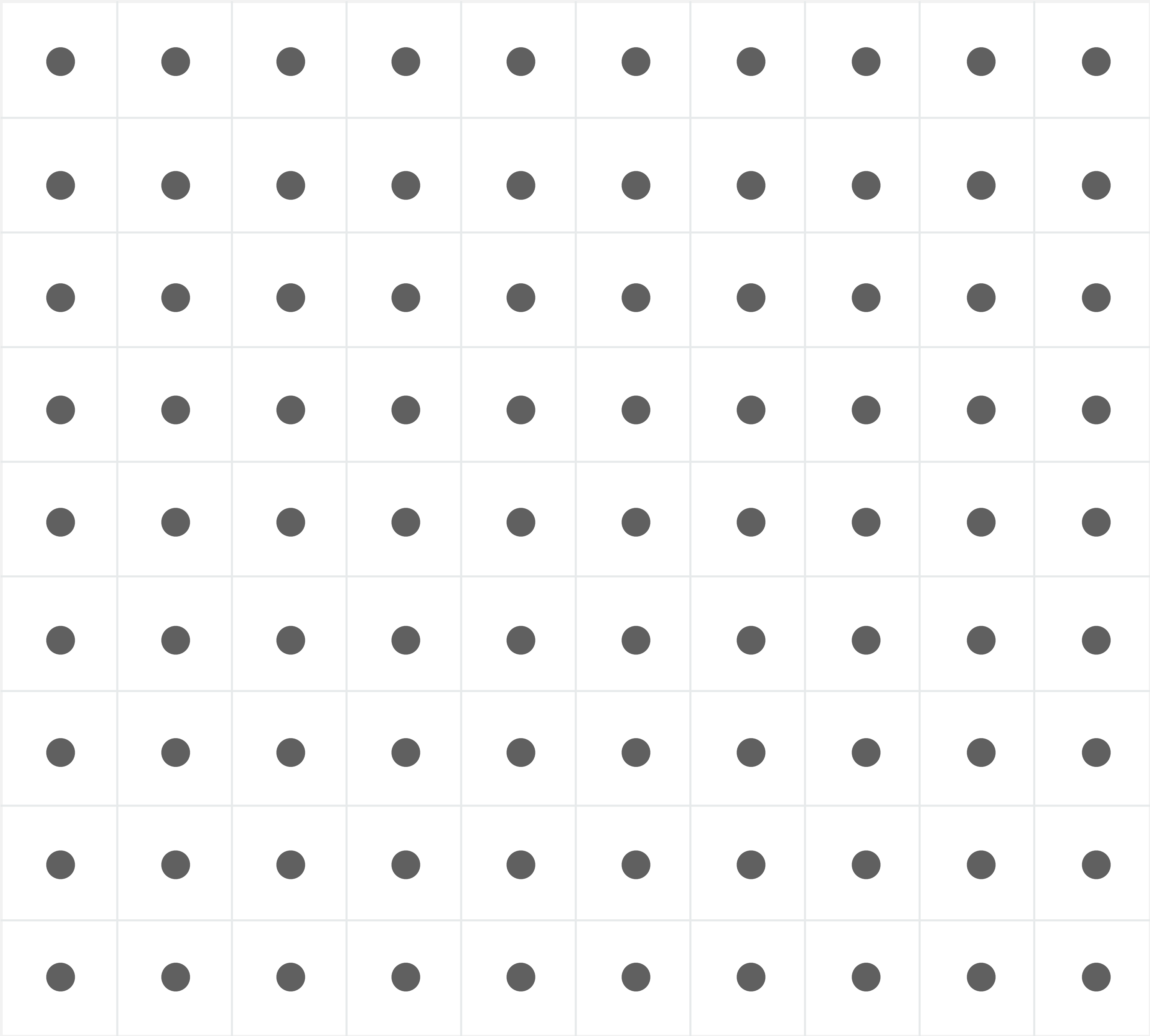
Seam carving. [Avidan–Shamir] Resize an image without distortion for display on cell phones and web browsers.

In the wild. Photoshop, Imagemagick, GIMP, ...

Content-aware resizing

To find vertical seam:

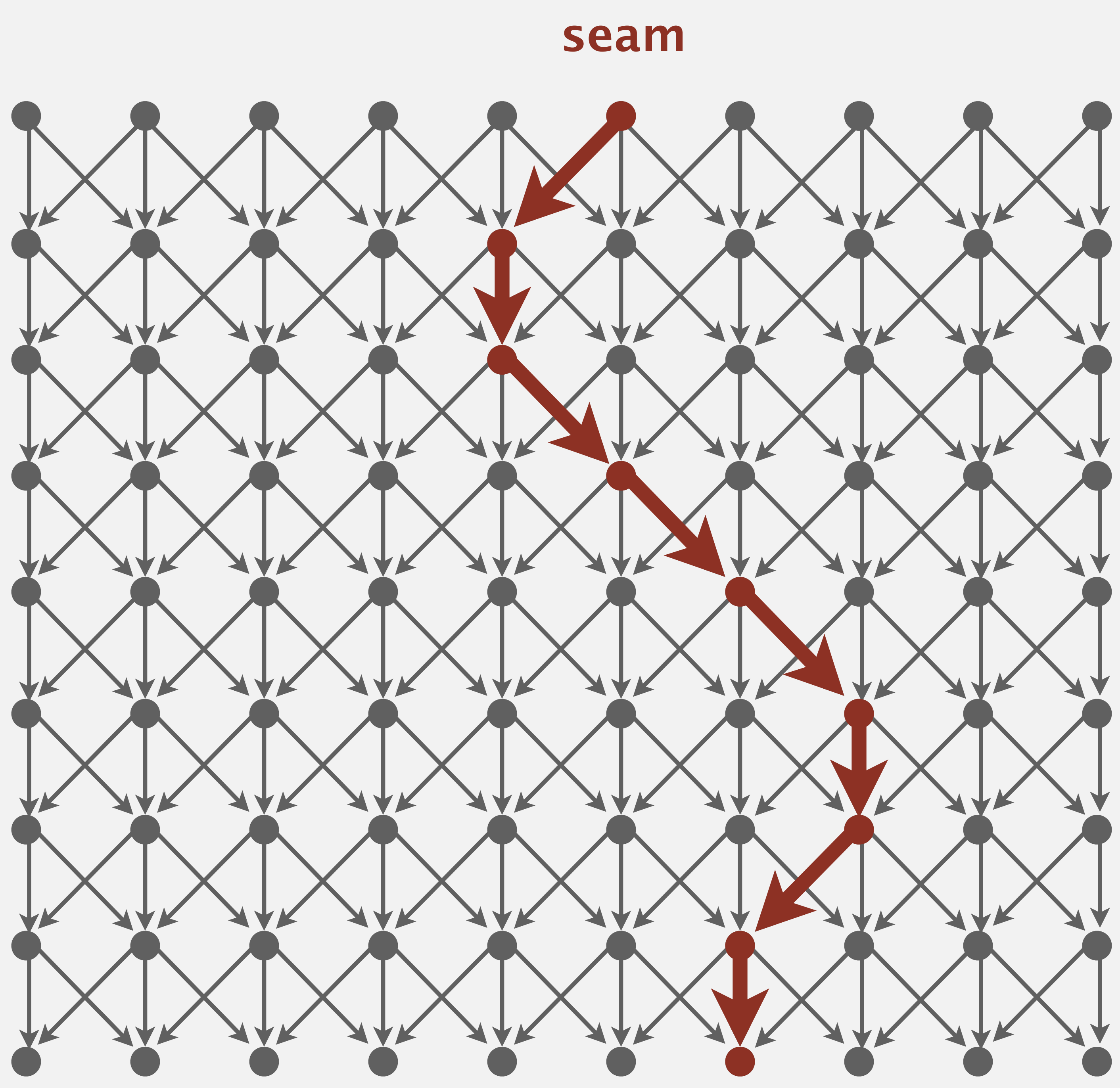
- Grid graph: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = “energy function” of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To find vertical seam:

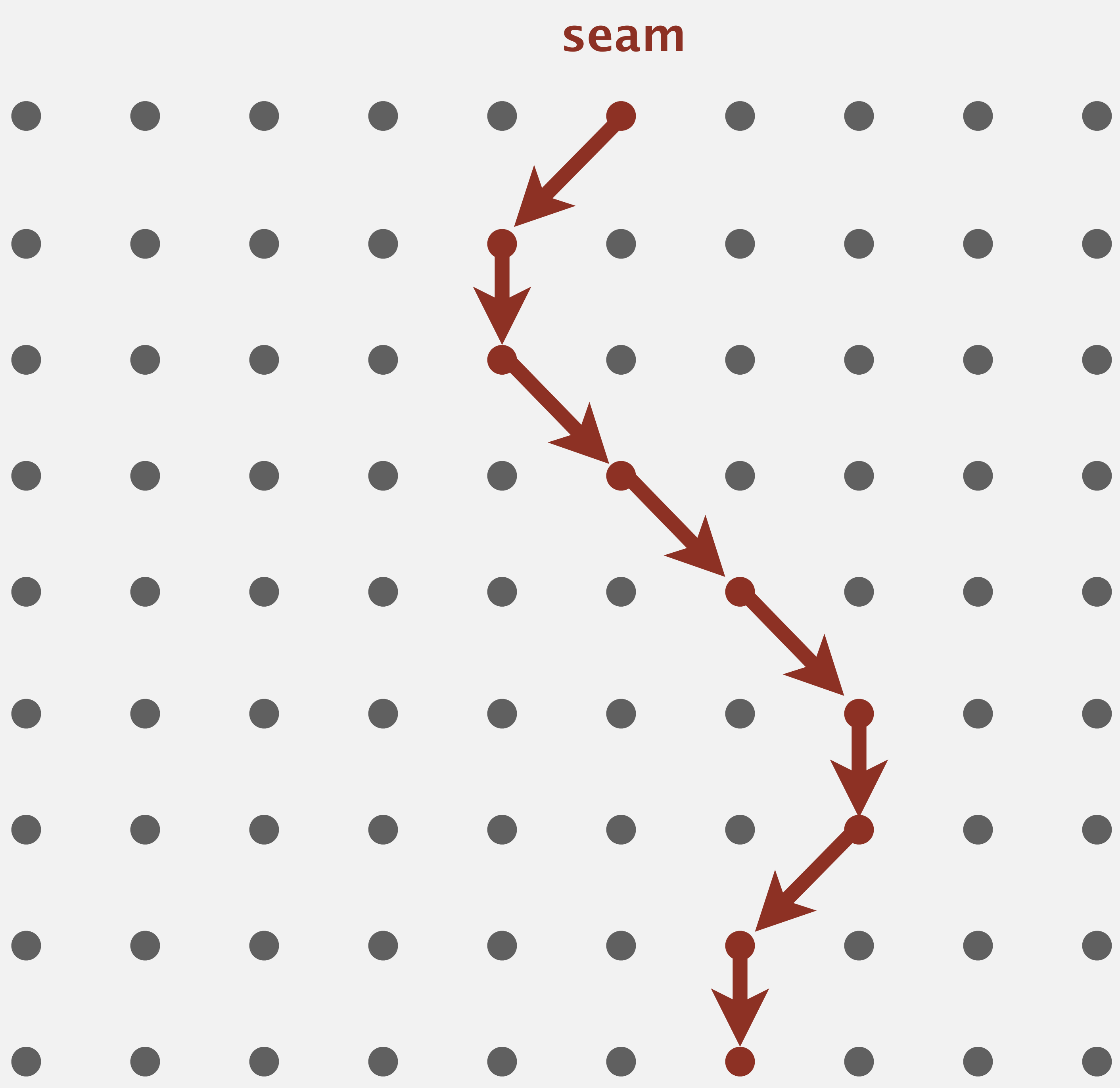
- Grid graph: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = "energy function" of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To remove vertical seam:

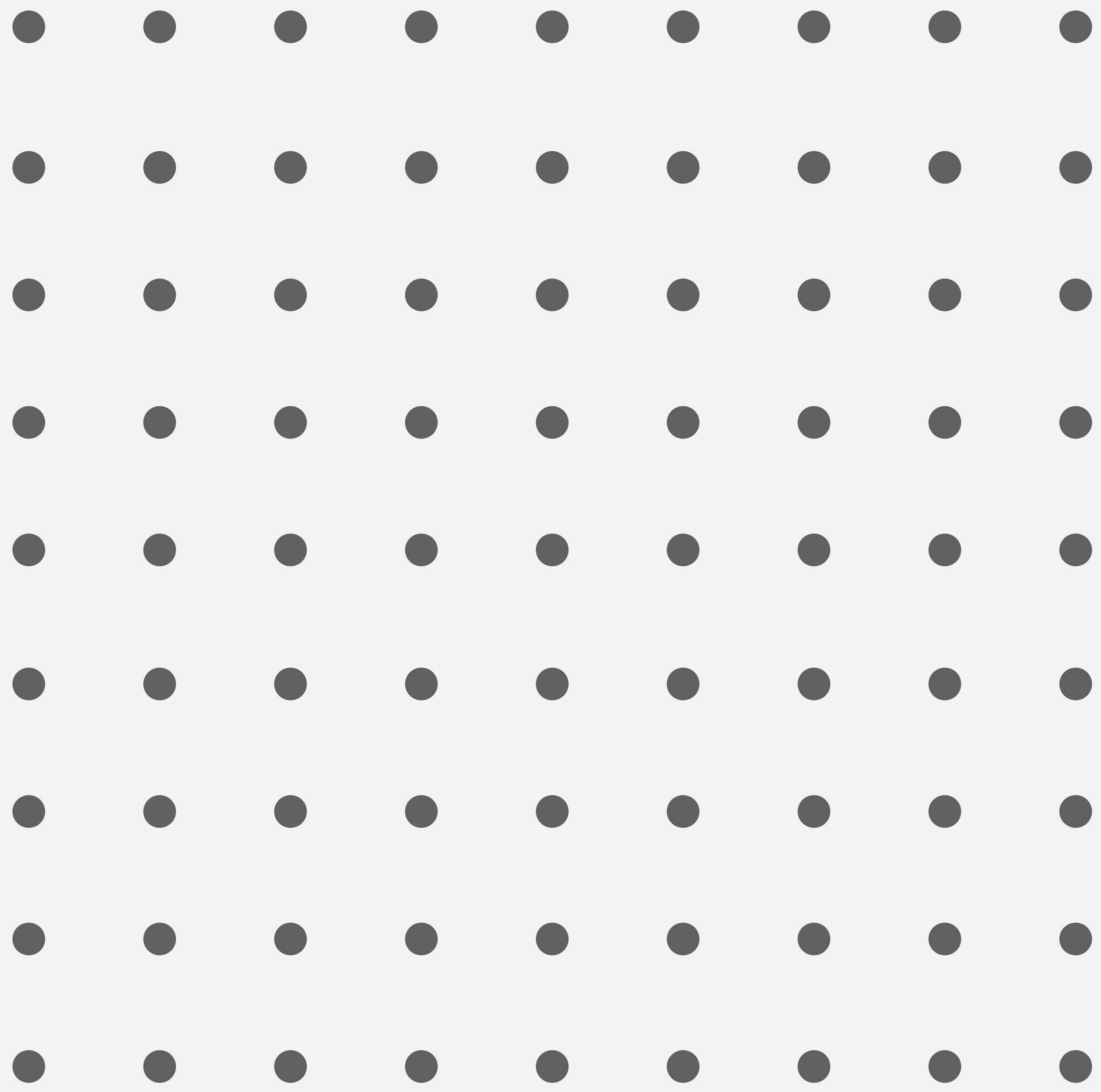
- Delete pixels on seam (one in each row).



Content-aware resizing

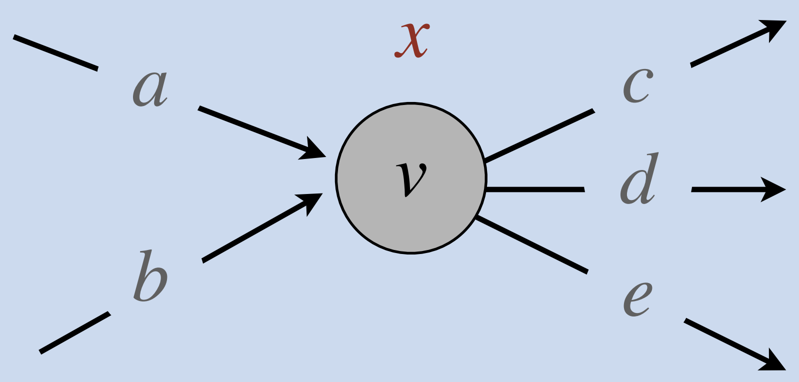
To remove vertical seam:

- Delete pixels on seam (one in each row).



Shortest Path Variants in a Digraph

Q1. How to model vertex weights (along with edge weights)?



Q2. How to model multiple sources and sinks?