Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 Shortest Paths

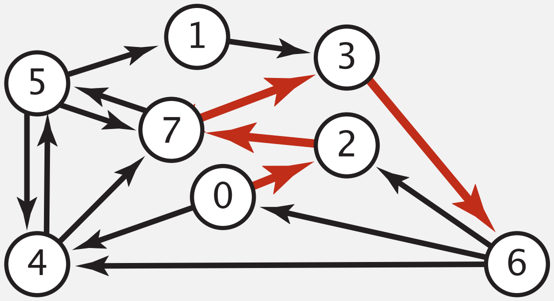
- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

Shortest paths in an edge-weighted digraph

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

edge-weighted digraph

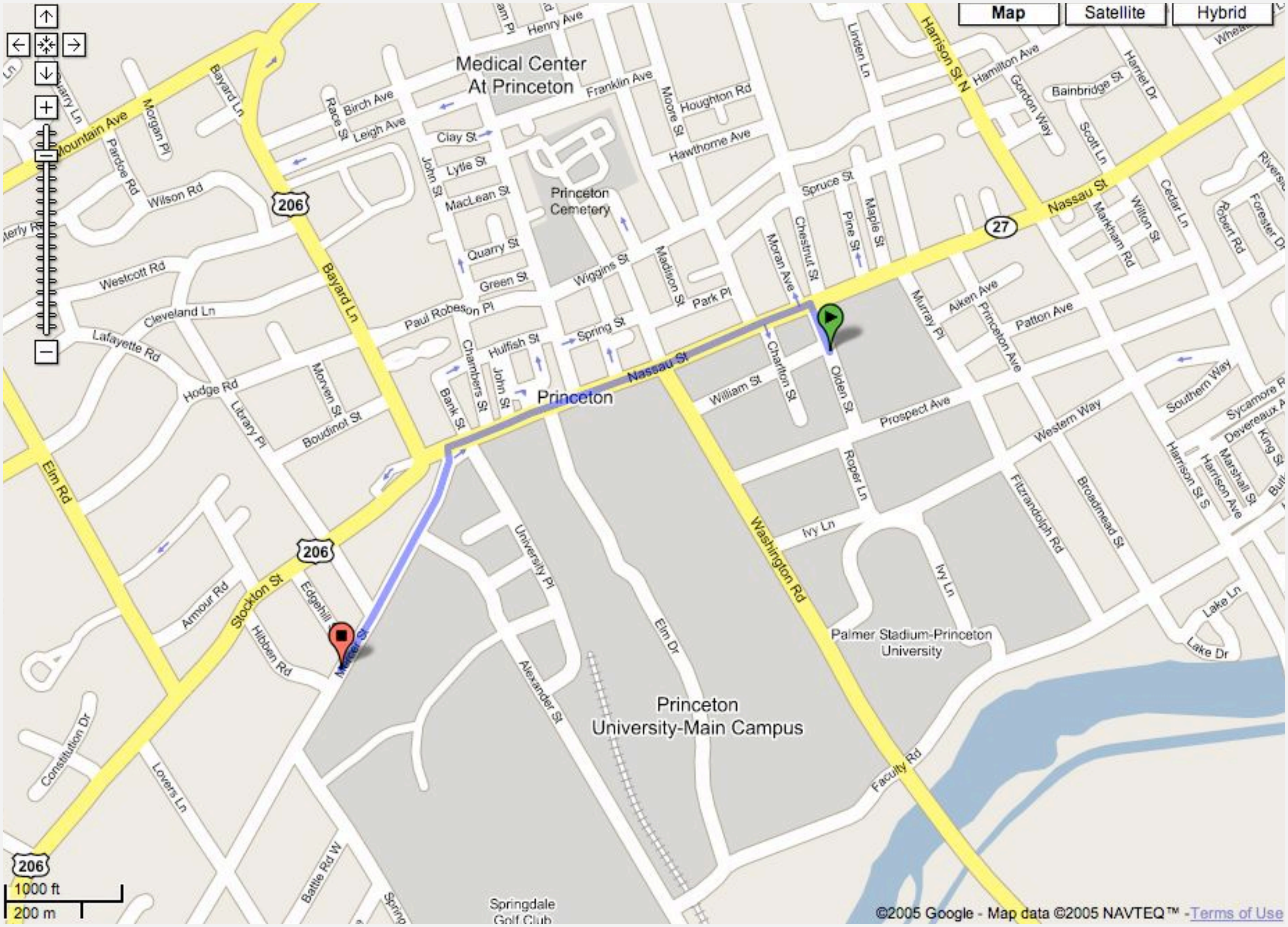
4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Google maps


Map
Satellite
Hybrid
Medical Center At Princeton
Princeton Cemetery
Princeton
Princeton University-Main Campus
Palmer Stadium-Princeton University
Springdale Golf Club
Nassau St
Washington Rd
Stockton St
Bayard Ln
Mountain Ave
Elm Rd
Harrison St N
Mercer St
Olden St
Alexander St
University Pl
Elm Dr
Faculty Rd
Prospect Ave
William St
Chambers St
Witherspoon St
Paul Robeson Pl
Hodge Rd
Library Pl
Edgehill
Hibben Rd
Lovers Ln
Battle Rd W
Constitution Dr
Armour Rd
Westcott Rd
Cleveland Ln
Lafayette Rd
Boudinot St
Morven St
Wilson Rd
Pardoe Rd
Morgan Pl
Birch Ave
Leigh Ave
Race St
Clay St
Lytle St
John St
MacLean St
Quarry St
Green St
Hulfish St
Spring St
Wiggins St
Bank St
Madison St
Park Pl
Franklin Ave
Henry Ave
Moore St
Houghton Rd
Hawthorne Ave
Spruce St
Linden Ln
Moran Ave
Chestnut St
Pine St
Maple St
Charlton St
Roper Ln
Ivy Ln
Murray Pl
Aiken Ave
Princeton Ave
Patton Ave
Western Way
Fitzrandolph Rd
Broadmead St
Southern Way
Hamilton Ave
Gordon Way
Bainbridge St
Harriet Dr
Scott Ln
Markham Rd
Wilton St
Cedar Ln
Robert Rd
Forester Dr
Sycamore Rd
Devereaux Ave
Harrison St S
Harrison Ave
Marshall St
King St
Lake Ln
Lake Dr
206
27
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

Car navigation

Shortest path applications

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

- 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 path variants

Which vertices?

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

Restrictions on edge weights?

- Nonnegative weights.
- Euclidean weights.
- Arbitrary weights.

Cycles?

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

Simplifying assumption. Shortest paths from s to each vertex v exist.

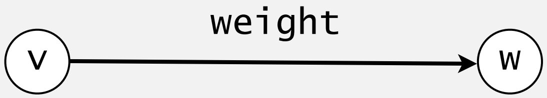
http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

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*
`String`	`toString()`	*string representation*



Idiom for processing an edge `e`: `int v = e.from(), w = e.to();`

Weighted directed edge: implementation in Java

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 int weight()
   {  return weight; }
}
```

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

Edge-weighted digraph API

`public class EdgeWeightedDigraph`

	`EdgeWeightedDigraph(int V)`	*edge-weighted digraph with V vertices*
	`EdgeWeightedDigraph(In in)`	*edge-weighted digraph from input stream*
`void`	`addEdge(DirectedEdge e)`	*add weighted directed edge e*
`Iterable<DirectedEdge>`	`adj(int v)`	*edges pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Iterable<DirectedEdge>`	`edges()`	*all edges*
`String`	`toString()`	*string representation*

Conventions. Allow self-loops and parallel edges.

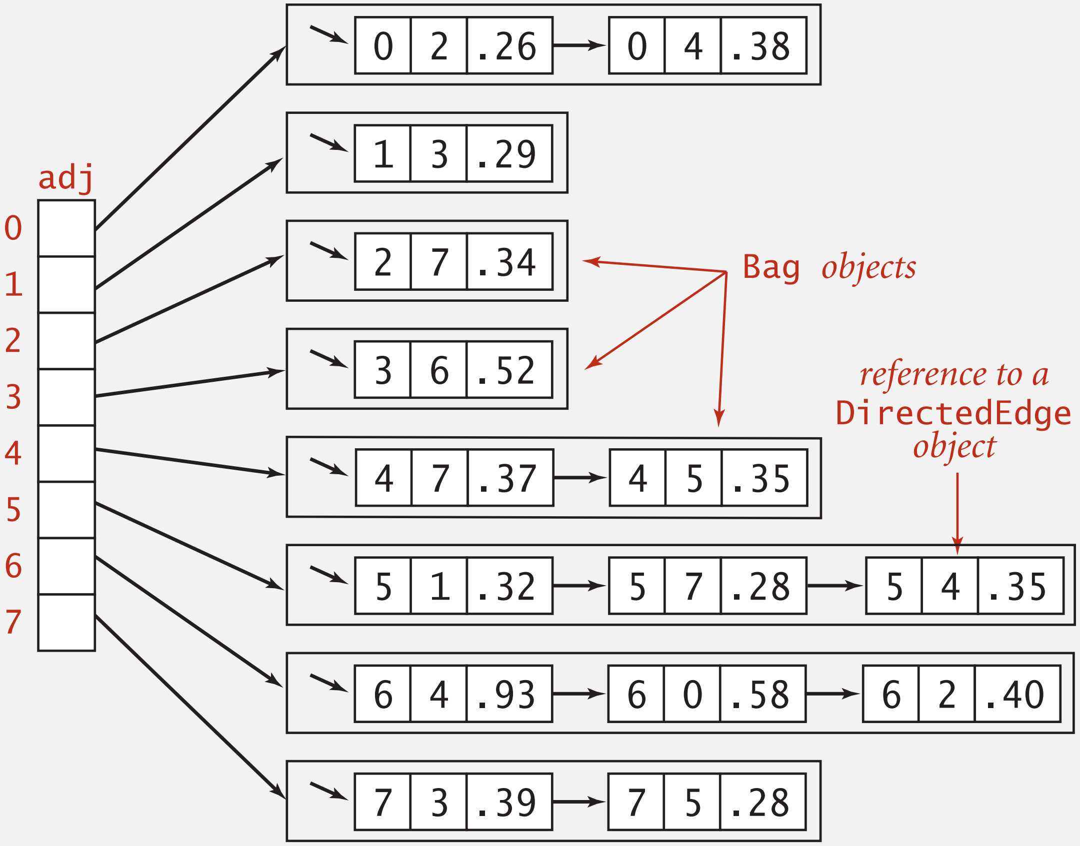
Edge-weighted digraph: adjacency-lists representation

`tinyEWD.txt`

V

E

```
8
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
```



`Bag` *objects*

reference to a `DirectedEdge` *object*

Edge-weighted digraph: adjacency-lists implementation in Java

Same as `EdgeWeightedGraph` except replace `Graph` with `Digraph`.

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

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

   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→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 graph 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?*

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
   StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
   for (DirectedEdge e : sp.pathTo(v))
      StdOut.print(e + "  ");
   StdOut.println();
}
```

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 graph 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?*

```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*


Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- APIs
- *shortest-paths properties*
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

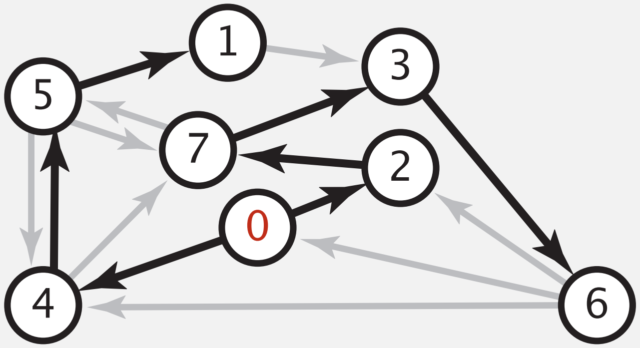
Data structures for single-source shortest paths

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

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

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

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



shortest-paths tree from 0

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

parent-link representation

Data structures for single-source shortest paths

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

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

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

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

```
public double distTo(int v)
{  return distTo[v];  }
```

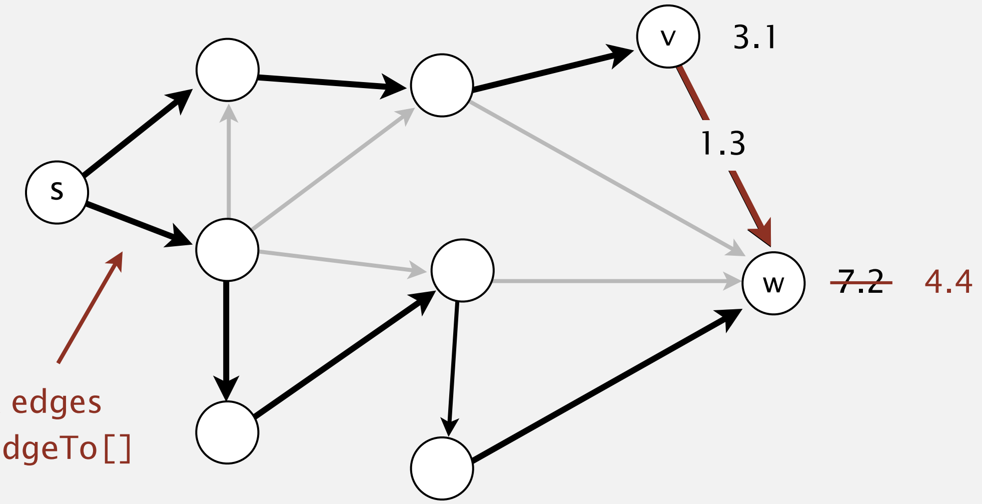
```
public Iterable<DirectedEdge> pathTo(int v)
{
   Stack<DirectedEdge> path = new Stack<DirectedEdge>();
   for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

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→w` gives shorter path to `w` through `v`, update both `distTo[w]` and `edgeTo[w]`.

v→w successfully relaxes



black edges are in edgeTo[]

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`→`w` gives shorter path to `w` through `v`,
  update both `distTo[w]` and `edgeTo[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;
   }
}
```

Shortest-paths optimality conditions

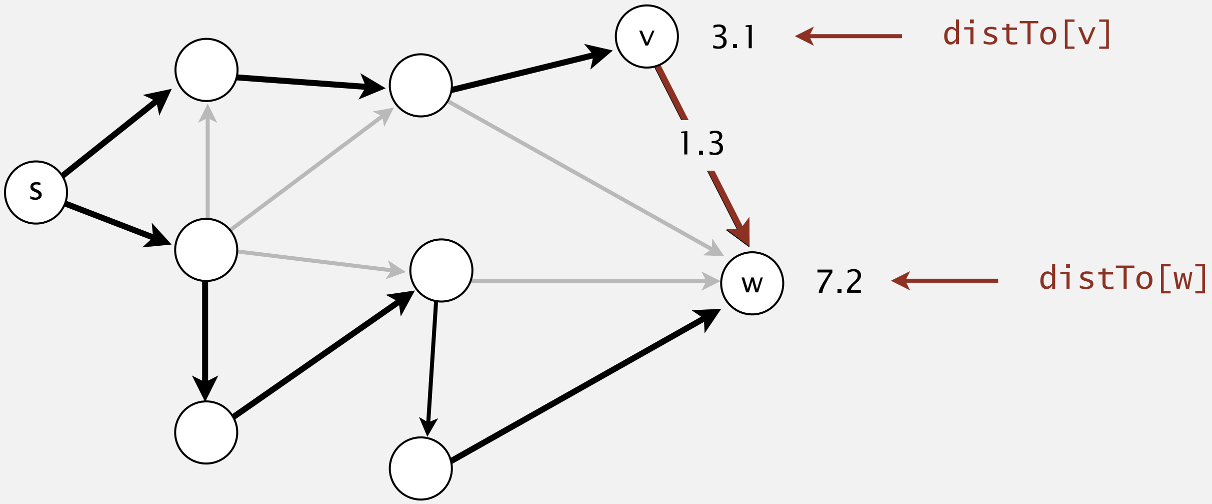
Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from `s` iff:

- For each vertex `v`, `distTo[v]` is the length of some path from `s` to `v`.
- For each edge `e` = `v`→`w`, `distTo[w]` ≤ `distTo[v] + e.weight()`.

Pf. ⇐ [necessary]

- Suppose that `distTo[w] > distTo[v] + e.weight()` for some edge `e` = `v`→`w`.
- Then, `e` gives a path from `s` to `w` (through `v`) of length less than `distTo[w]`.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.
Then `distTo[]` are the shortest path distances from `s` iff:

- For each vertex `v`, `distTo[v]` is the length of some path from `s` to `v`.
- For each edge `e` = `v`→`w`, `distTo[w]` ≤ `distTo[v]` + `e.weight()`.

Pf. ⇒ [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from s to w.
- Then,

$$\begin{aligned}
\texttt{distTo}[v_1] &\le \texttt{distTo}[v_0] + e_1.\texttt{weight()} \\
\texttt{distTo}[v_2] &\le \texttt{distTo}[v_1] + e_2.\texttt{weight()} \\
\ldots \\
\texttt{distTo}[v_k] &\le \texttt{distTo}[v_{k-1}] + e_k.\texttt{weight()}
\end{aligned}$$

e_i = i^{th} edge on shortest path from s to w

- Add inequalities; simplify; and substitute `distTo[v₀] = distTo[s] = 0`:

$$\texttt{distTo}[w] = \texttt{distTo}[v_k] \le \underbrace{e_1.\texttt{weight()} + e_2.\texttt{weight()} + \ldots + e_k.\texttt{weight()}}_{\text{weight of shortest path from s to w}}$$

- Thus, `distTo[w]` is the weight of shortest path to w. ▪

weight of some path from s to w

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

  - **Relax any edge.**

Proposition. Generic algorithm computes SPT (if it exists) from `s`.

Pf sketch.

- Throughout algorithm, `distTo[v]` is the length of a simple path from `s` to v (and `edgeTo[v]` is last edge on path).
- Each successful relaxation decreases `distTo[v]` for some `v`.
- The entry `distTo[v]` can decrease at most a finite number of times. ▪

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- ***shortest-paths properties***
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- APIs
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

Edsger W. Dijkstra: select quotes

“ Do only what only you can do. ”

“ In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind. ”

“ The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. ”

“ It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. ”

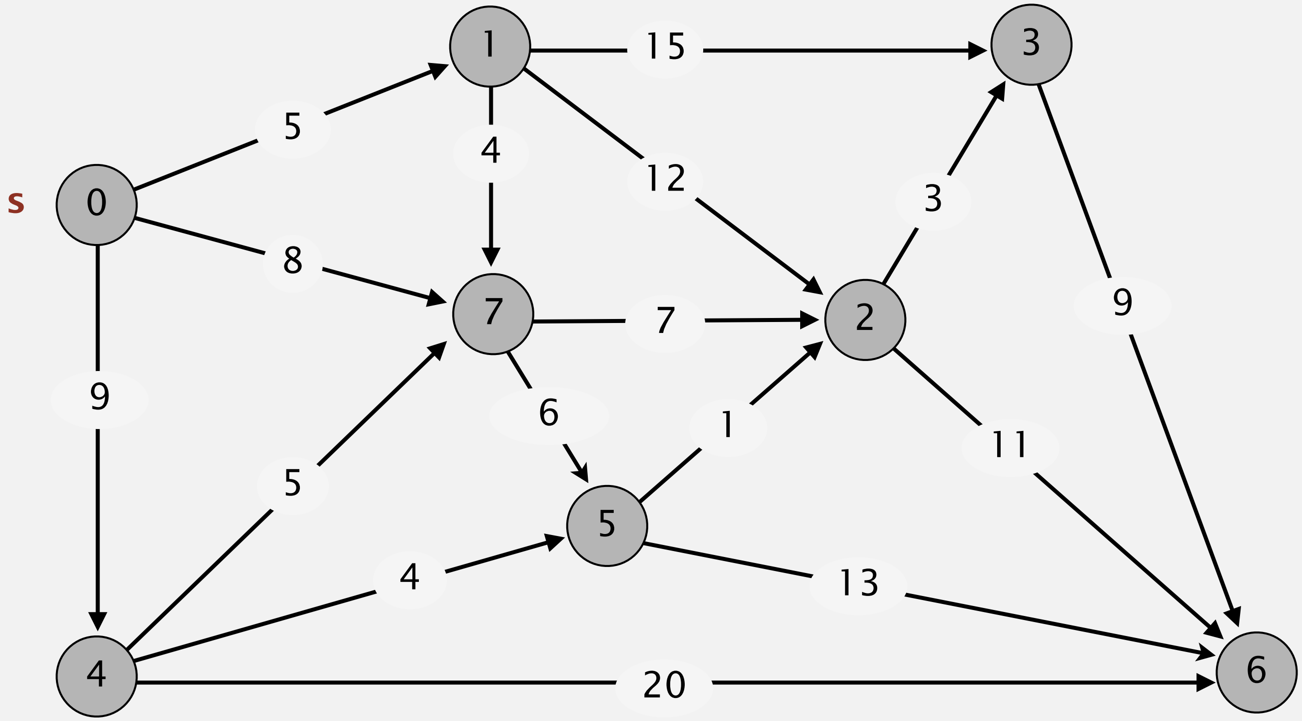
“ APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums. ”

Edsger W. Dijkstra
Turing award 1972

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 pointing from that vertex.

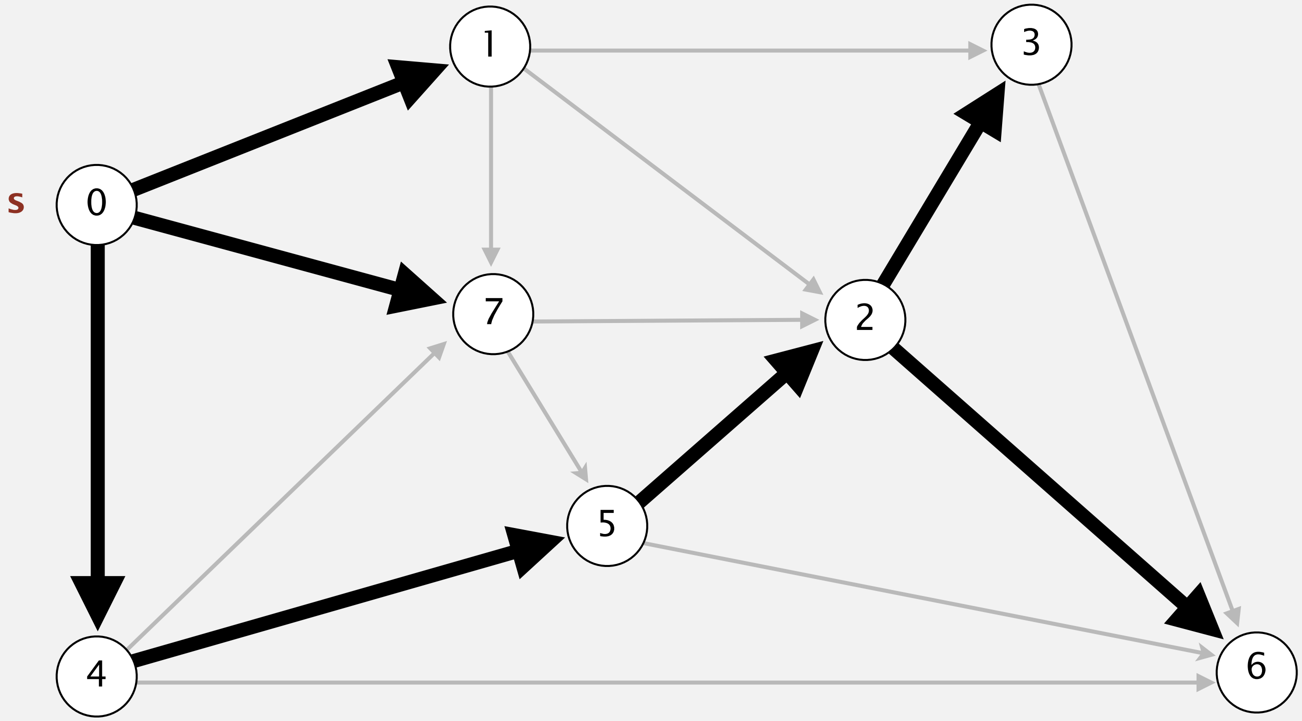


an edge-weighted digraph

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

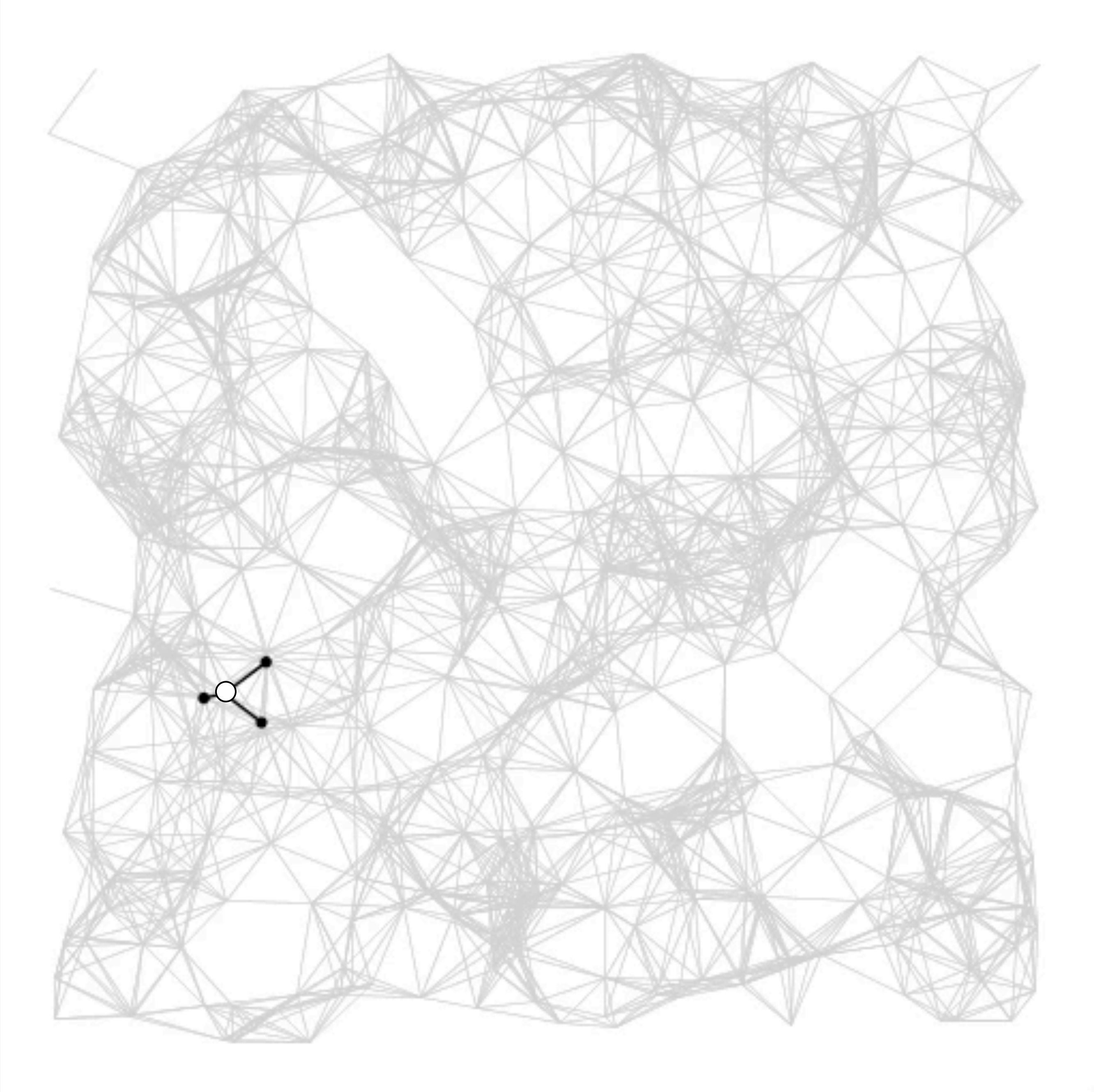
- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



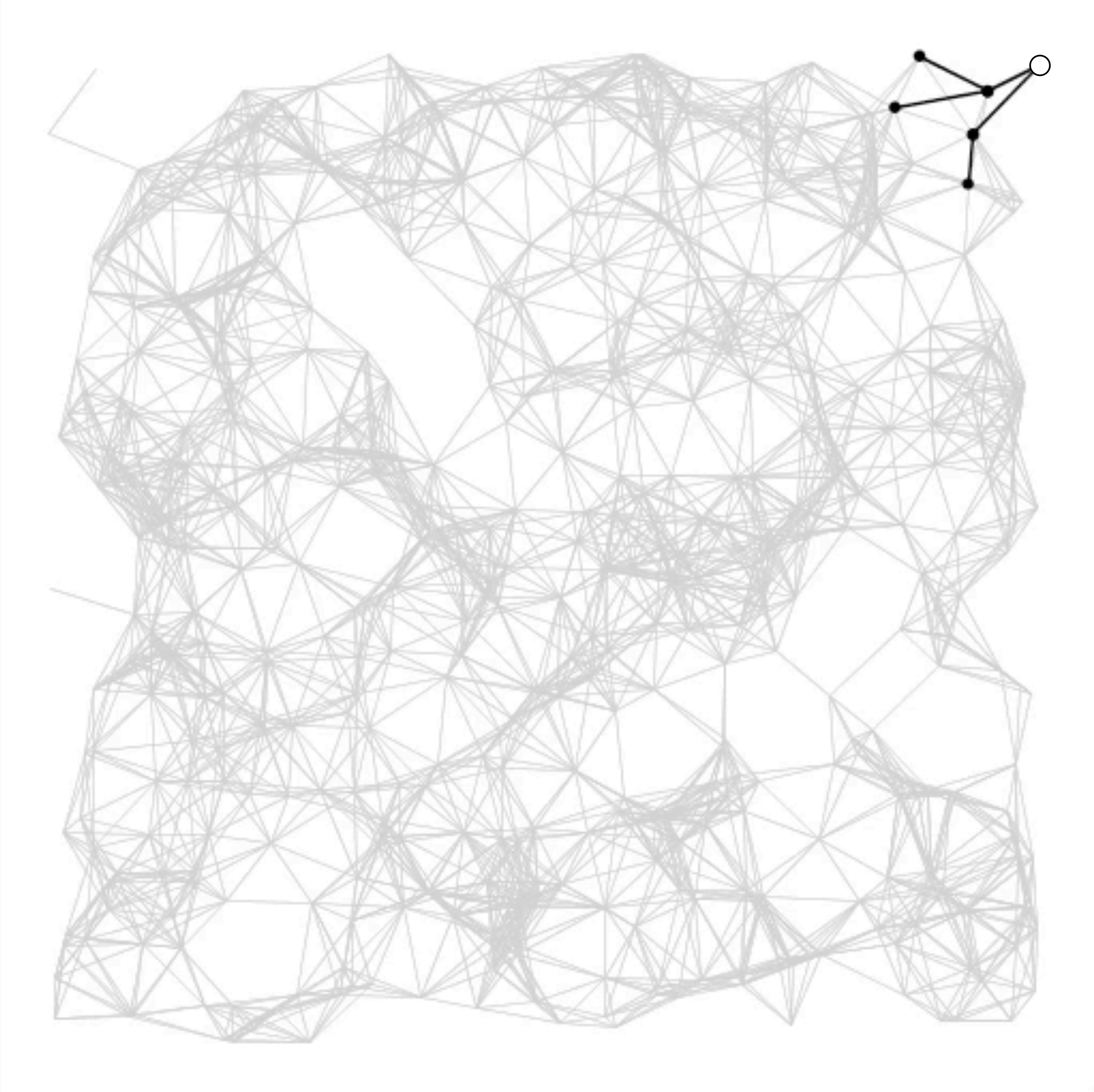
v	`distTo[]`	`edgeTo[]`
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Dijkstra's algorithm visualization



Dijkstra's algorithm visualization



Dijkstra's algorithm: correctness proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge `e` = `v`→`w` is relaxed exactly once (when `v` is relaxed), leaving `distTo[w] ≤ distTo[v] + e.weight()`.
- Inequality holds until algorithm terminates because:
  - `distTo[w]` cannot increase ← `distTo[]` values are monotone decreasing
  - `distTo[v]` will not change ← we choose lowest `distTo[]` value at each step (and edge weights are nonnegative)
- Thus, upon termination, shortest-paths optimality conditions hold. ▪

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);
      }
    }
 }
```

relax vertices in order
of distance from s

Dijkstra's algorithm: Java implementation

```
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.decreaseKey(w, distTo[w]);
       else                pq.insert     (w, distTo[w]);
   }
}
```

update PQ

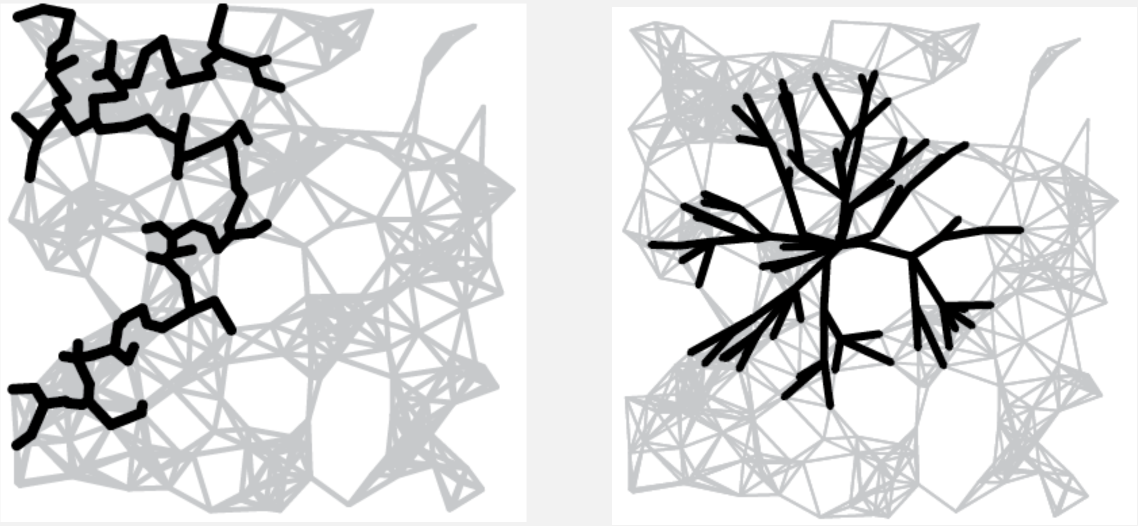
Computing spanning trees in graphs

Dijkstra's algorithm seem familiar?

- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a graph's spanning tree.

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

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



Note: DFS and BFS are also in this family of algorithms.

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, 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 (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

Bottom line.

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

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- APIs
- shortest-paths properties
- **Dijkstra's algorithm**
- edge-weighted DAGs
- negative weights

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

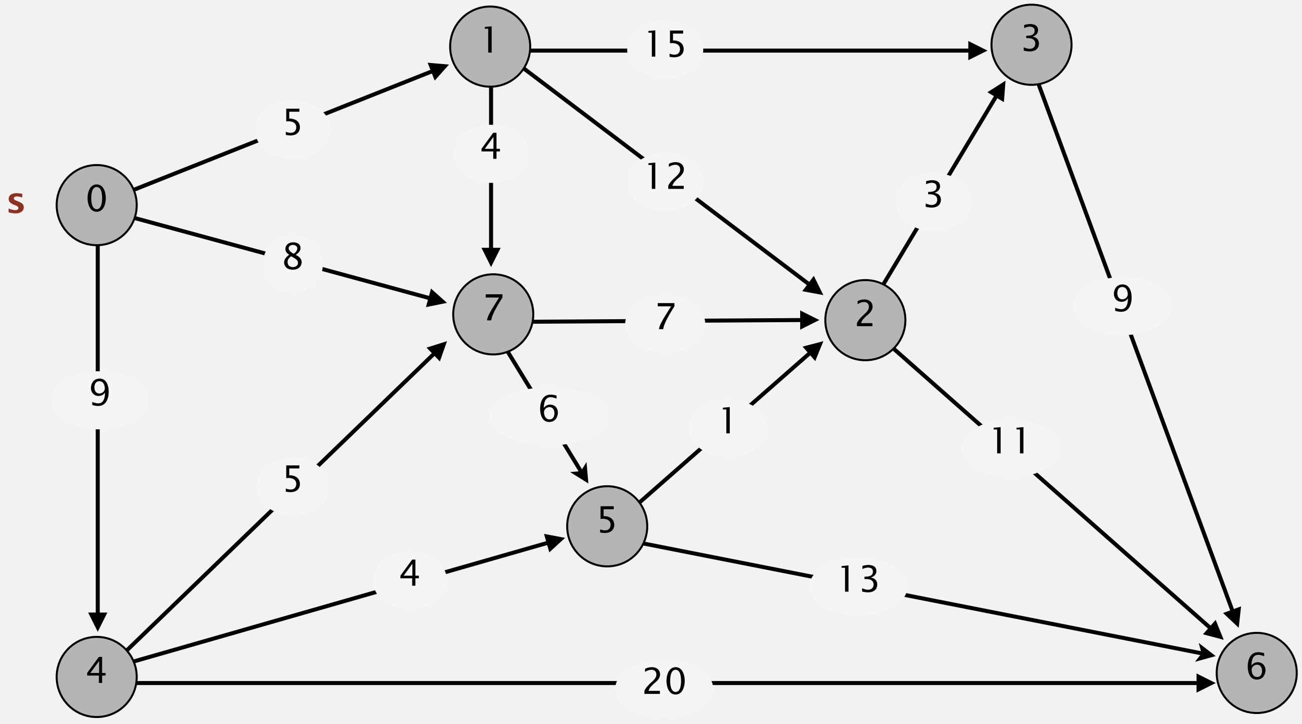
Acyclic edge-weighted digraphs

Q. Suppose that an edge-weighted digraph has no directed cycles.
Is it easier to find shortest paths than in a general digraph?

A. Yes!

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

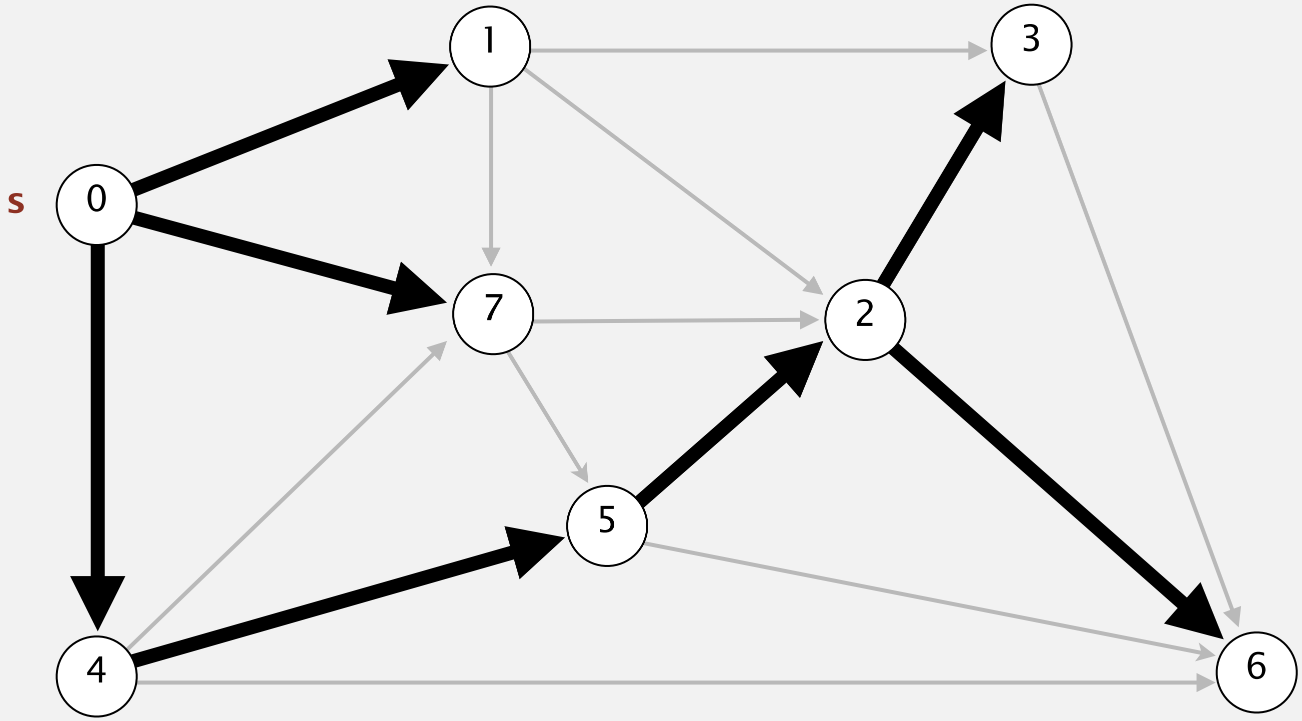


an edge-weighted DAG

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

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



0 1 4 7 5 2 3 6

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

Shortest paths in edge-weighted DAGs

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

edge weights can be negative!

Pf.

- Each edge `e` = `v`→`w` is relaxed exactly once (when `v` is relaxed), leaving `distTo[w] ≤ distTo[v] + e.weight()`.
- Inequality holds until algorithm terminates because:
  - `distTo[w]` cannot increase ← `distTo[]` values are monotone decreasing
  - `distTo[v]` will not change ← because of topological order, no edge pointing to v will be relaxed after v is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold. ▪

Shortest paths in edge-weighted DAGs

```
public class AcyclicSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;

   public AcyclicSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];

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

      Topological topological = new Topological(G);
      for (int v : topological.order())
         for (DirectedEdge e : G.adj(v))
            relax(e);
    }
 }
```

topological order

Content-aware resizing

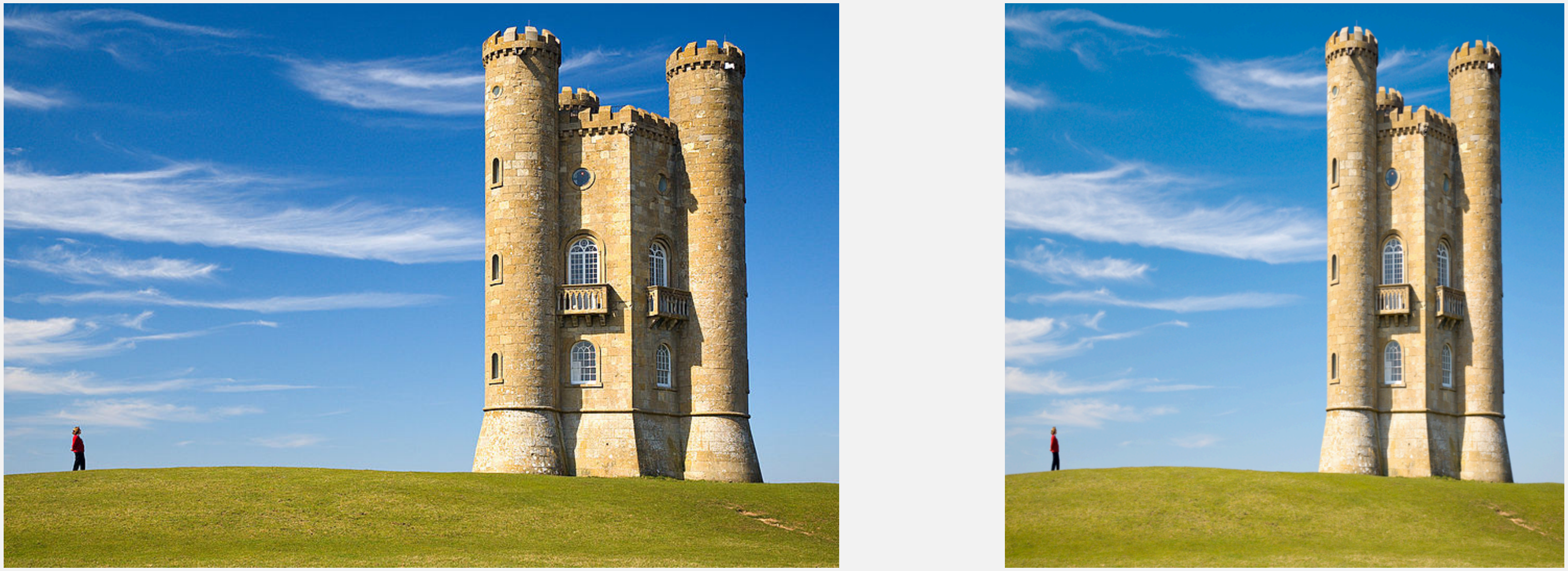
Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.

Shai Avidan
Mitsubishi Electric Research Lab
Ariel Shamir
The interdisciplinary Center & MERL

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

Content-aware resizing

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

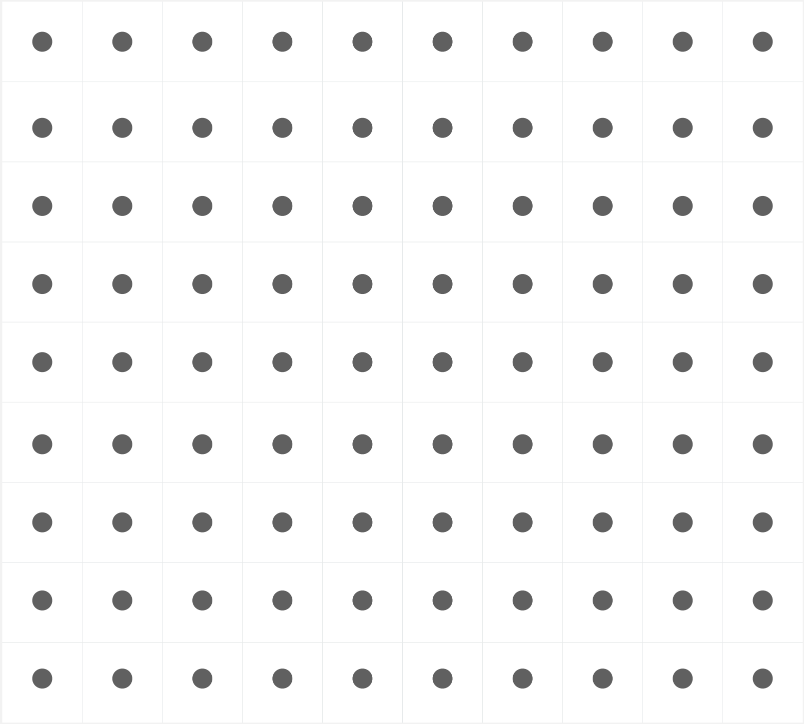


In the wild. Photoshop CS 5, Imagemagick, GIMP, ...

Content-aware resizing

To find vertical seam:

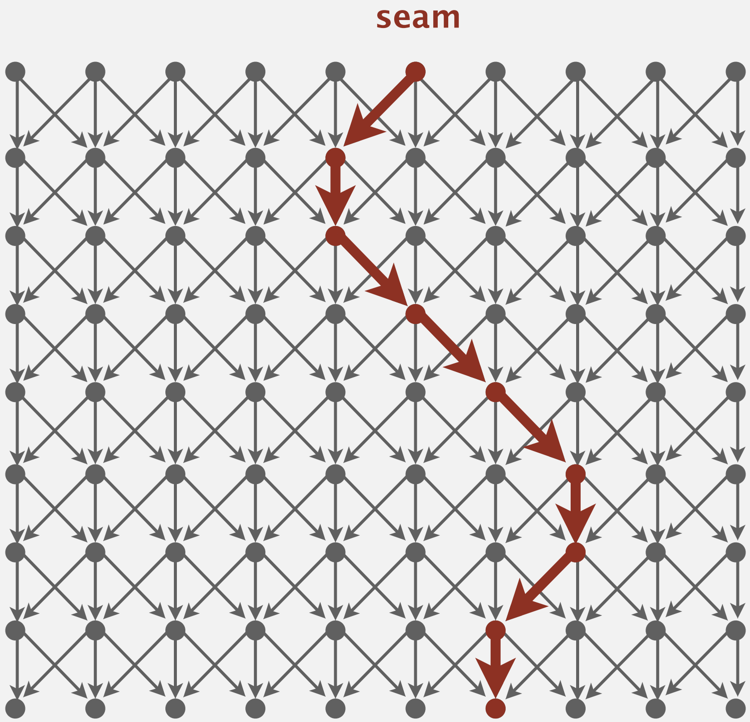
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To find vertical seam:

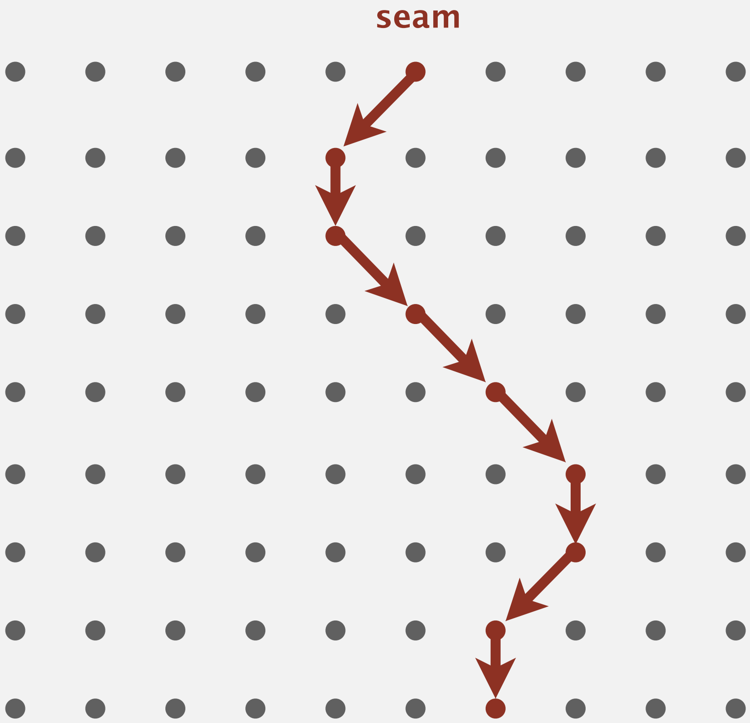
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To remove vertical seam:

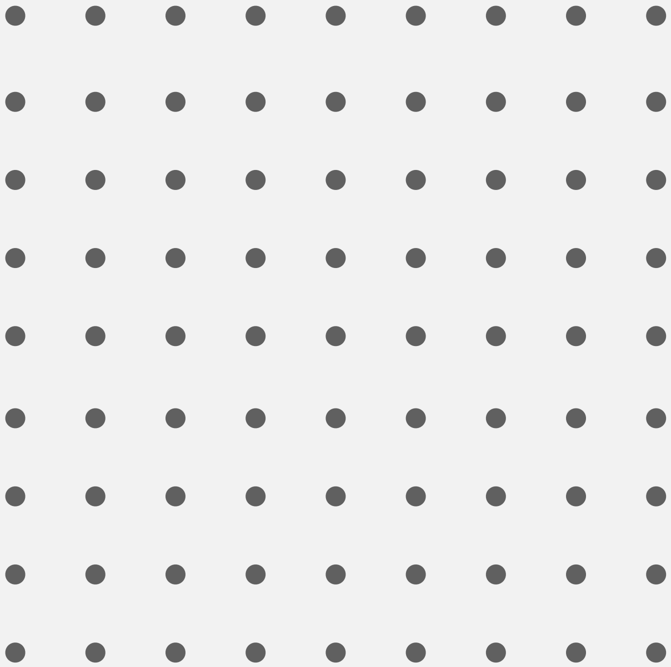
- Delete pixels on seam (one in each row).



Content-aware resizing

To remove vertical seam:

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



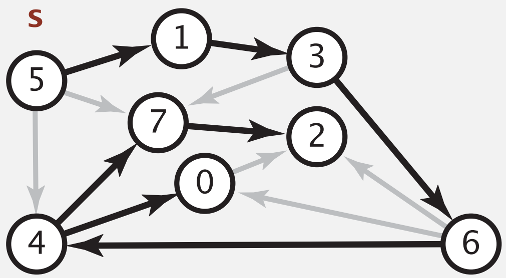
Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.

equivalent: reverse sense of equality in `relax()`

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

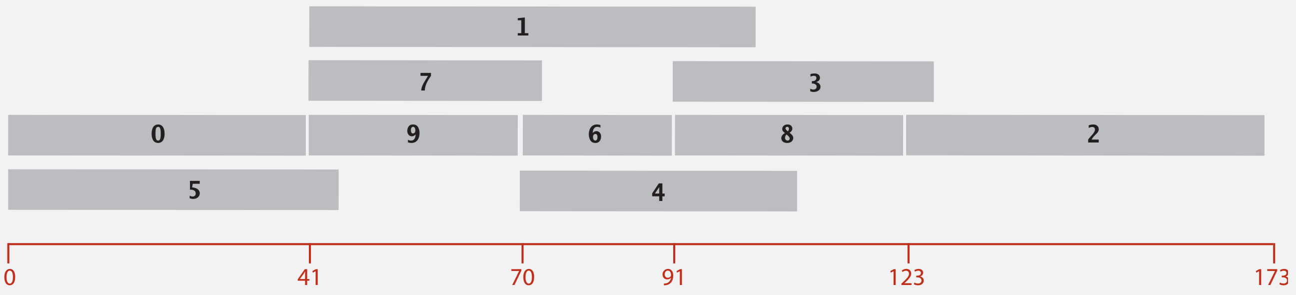


Key point. Topological sort algorithm works even with negative weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6



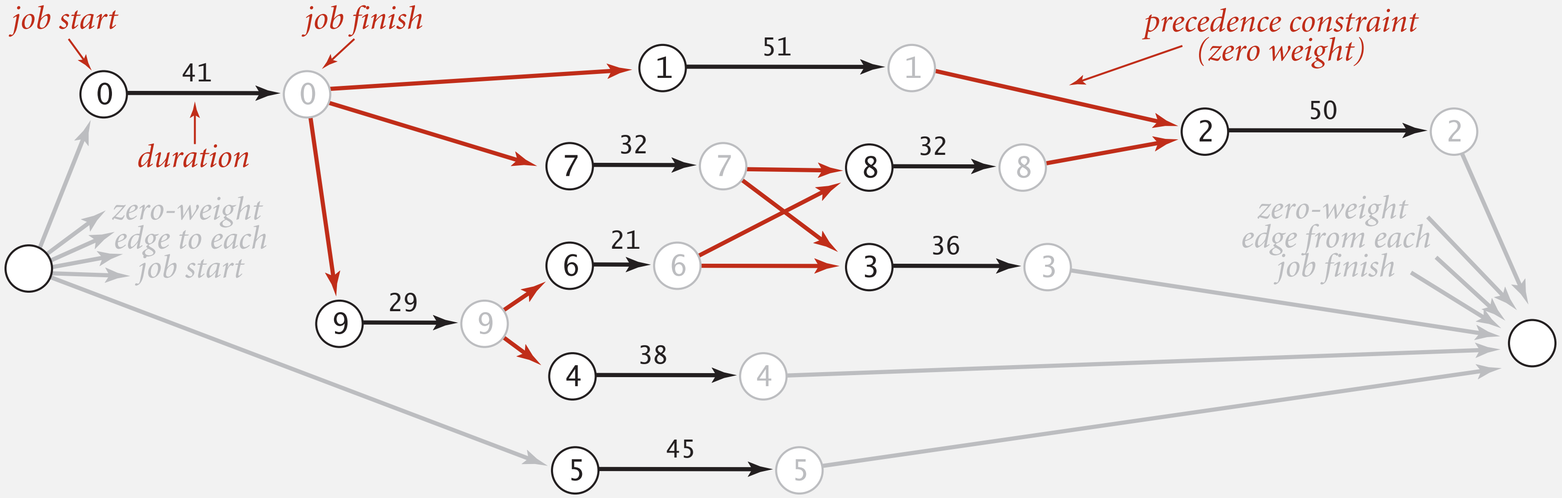
Parallel job scheduling solution

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

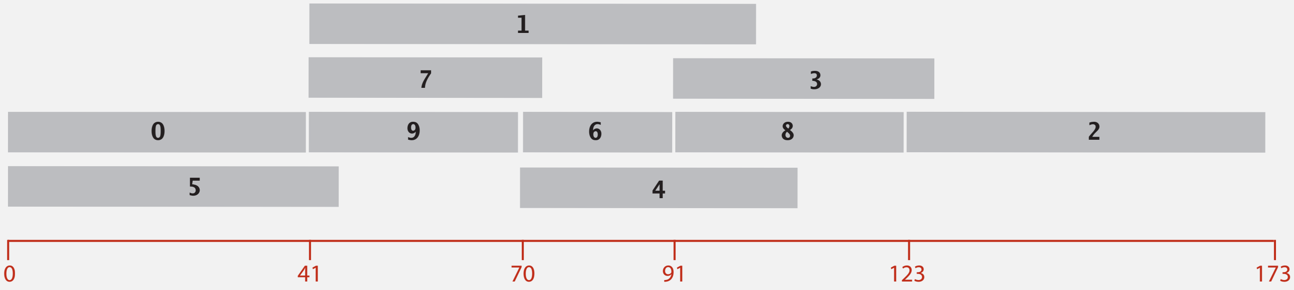
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6

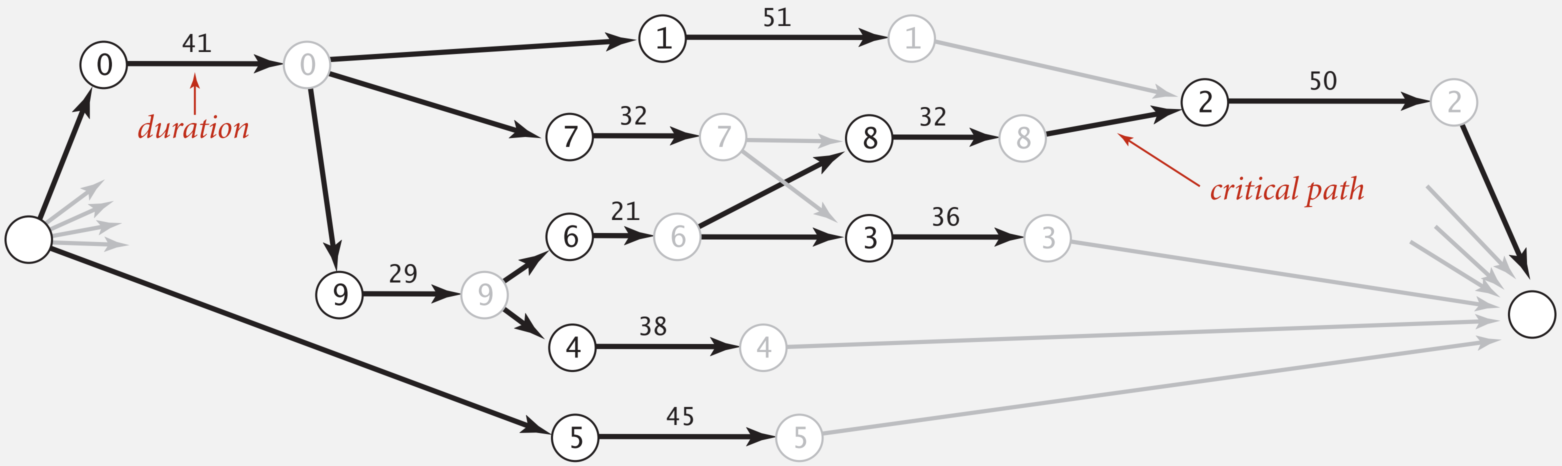


Critical path method

CPM. Use longest path from the source to schedule each job.



Parallel job scheduling solution



Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

Algorithms

Robert Sedgewick | Kevin Wayne

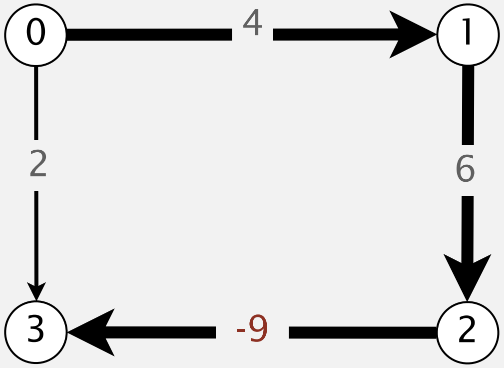
http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

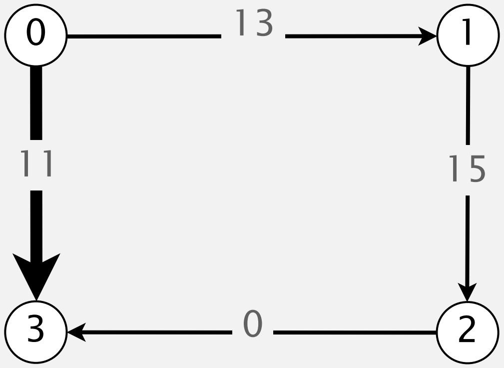
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the
shortest path from 0→1→2→3 to 0→3.

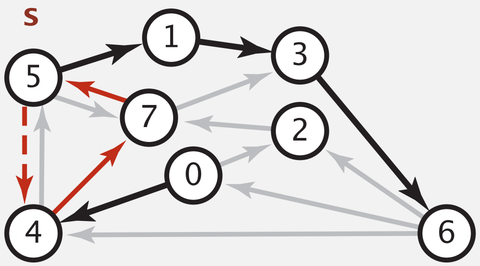
Conclusion. Need a different algorithm.

Negative cycles

Def. A negative cycle is a directed cycle whose sum of edge weights is negative.

digraph

4->5	0.35
5->4	-0.66
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



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Bellman-Ford algorithm

Bellman–Ford algorithm

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat V times:

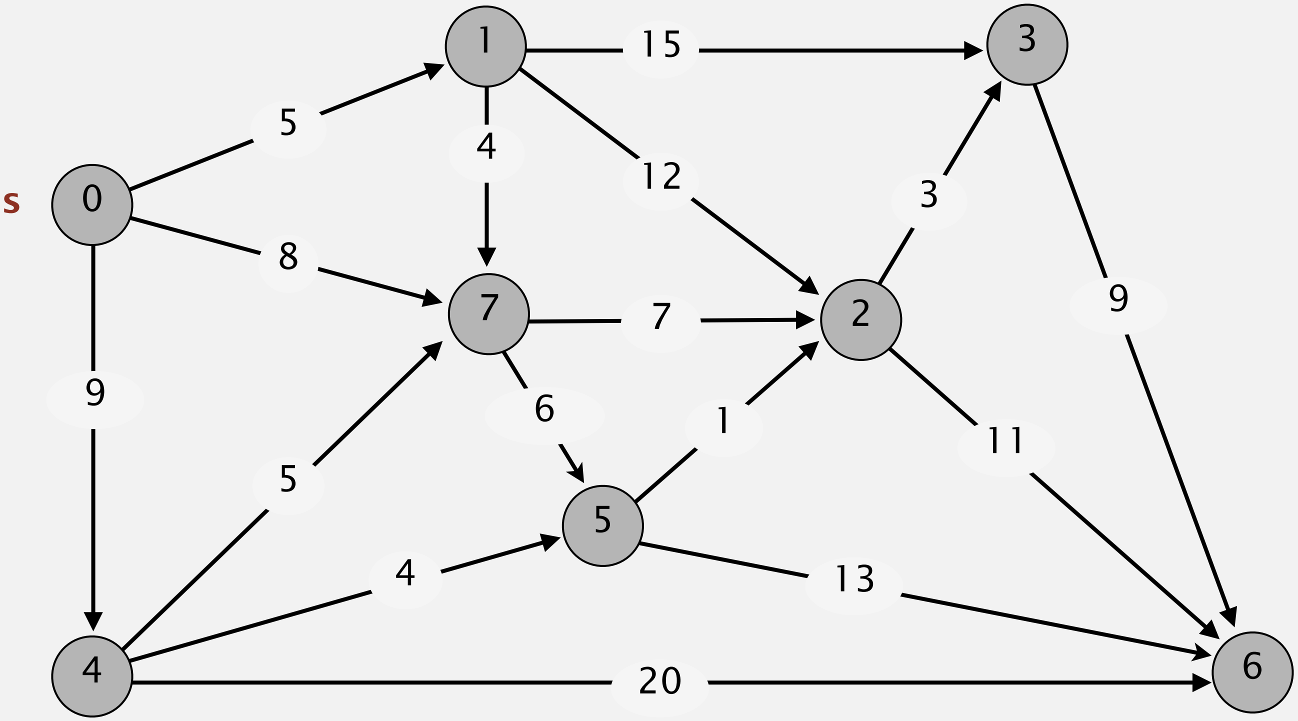
- **Relax each edge.**

```
for (int i = 0; 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)

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

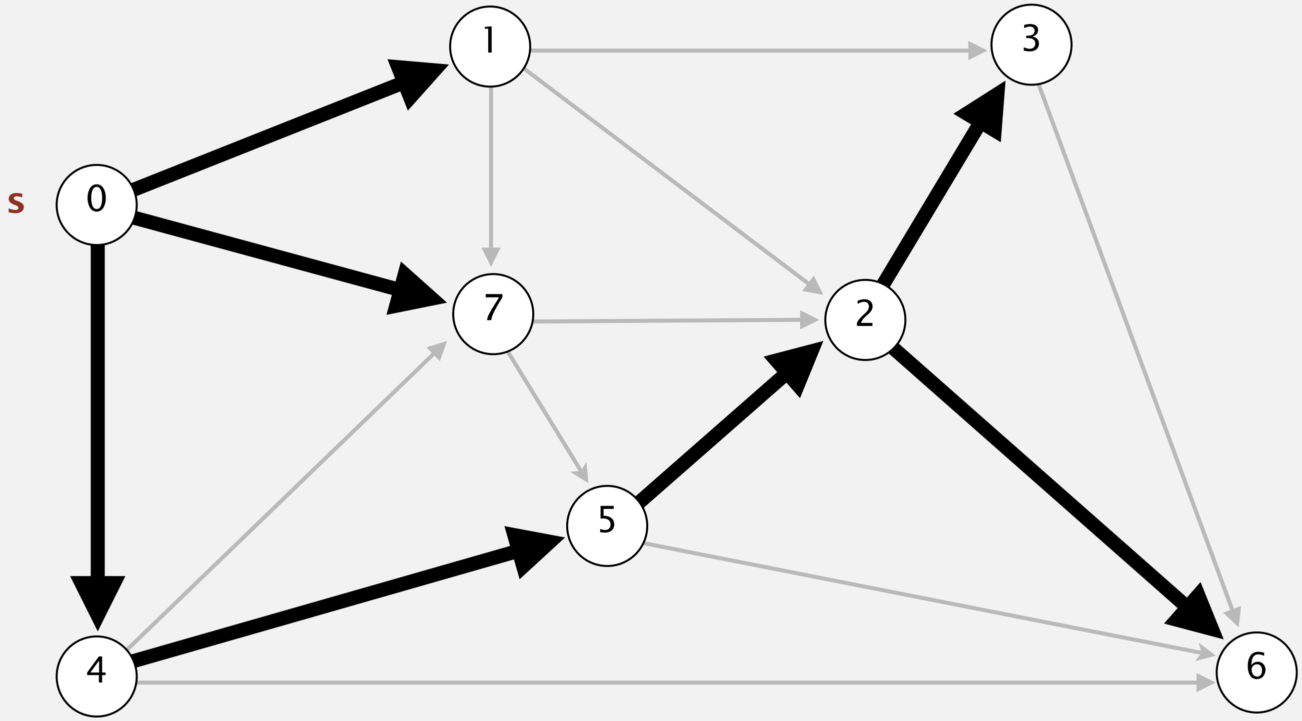


an edge-weighted digraph

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 demo

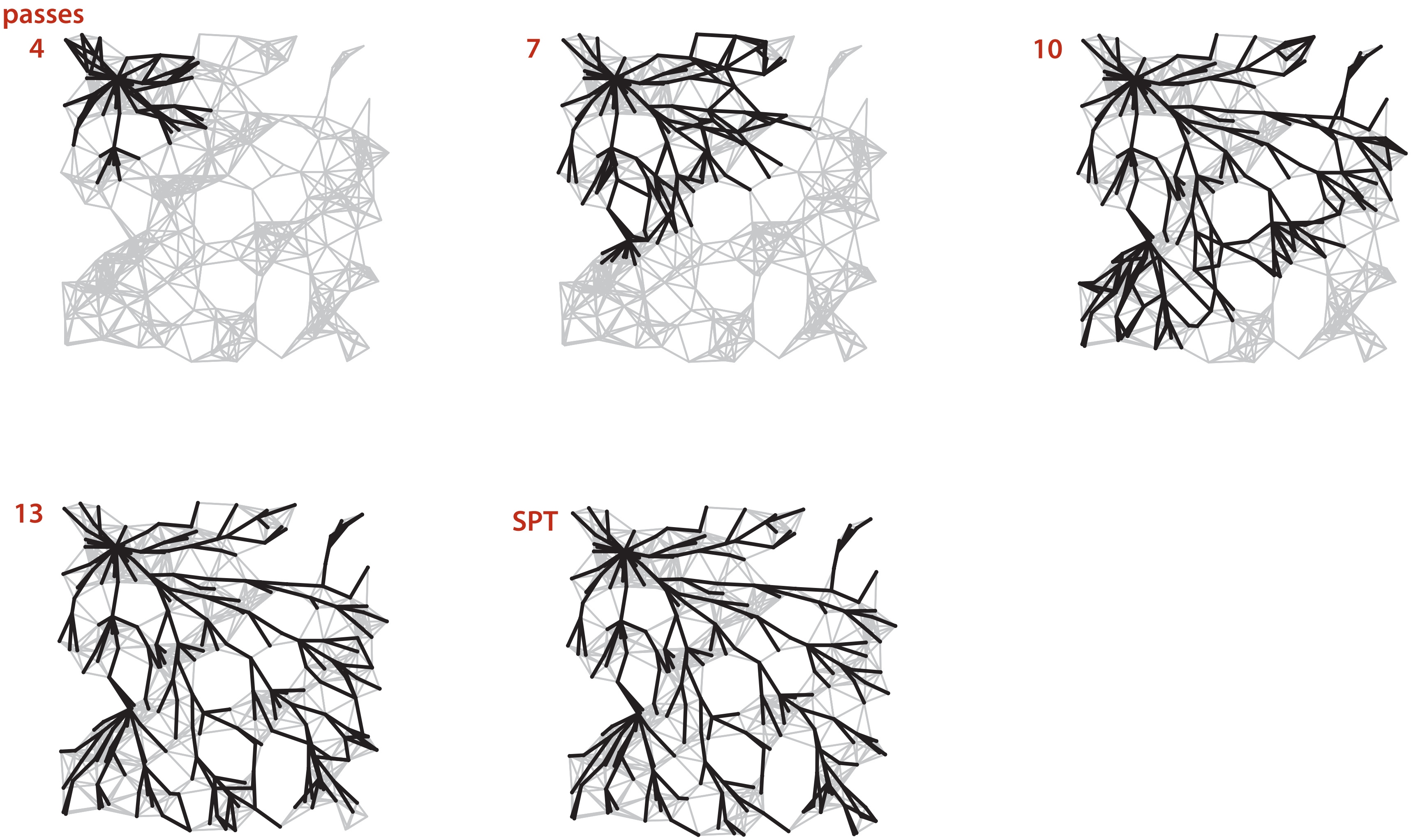
Repeat V times: relax all E edges.



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

shortest-paths tree from vertex s

Bellman-Ford algorithm visualization



Bellman-Ford algorithm: analysis

Bellman–Ford algorithm

```
Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat V times:
    – Relax each edge.
```

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass i, found shortest path containing at most i edges.

Bellman-Ford algorithm: practical improvement

Observation. If `distTo[v]` does not change during pass `i`, no need to relax any edge pointing from `v` in pass `i+1`.

FIFO implementation. Maintain queue of vertices whose `distTo[]` changed.

be careful to keep at most one copy of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

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.

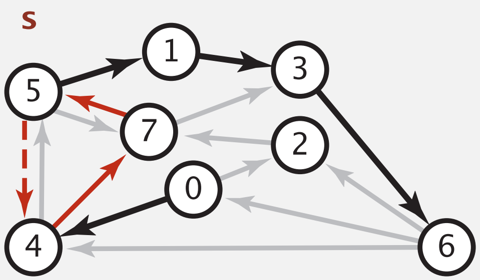
Finding a negative cycle

Negative cycle. Add two method to the API for `SP`.

`boolean`	`hasNegativeCycle()`	*is there a negative cycle?*
`Iterable <DirectedEdge>`	`negativeCycle()`	*negative cycle reachable from s*

digraph

```
4->5  0.35
5->4 -0.66
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
```

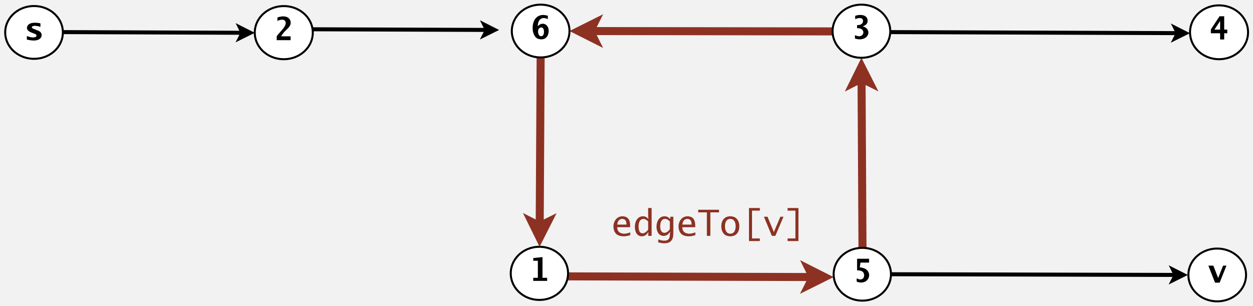


negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

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.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

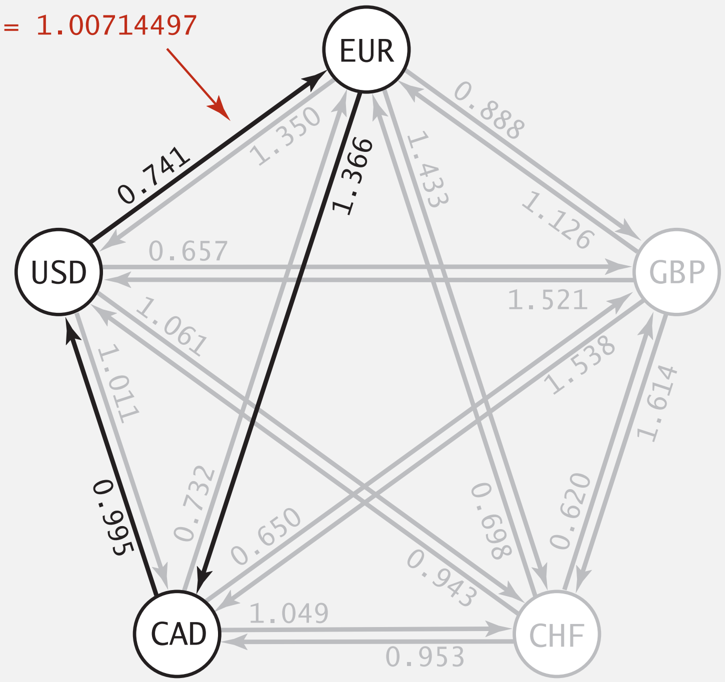
Ex. \$1,000 ⇒ 741 Euros ⇒ 1,012.206 Canadian dollars ⇒ \$1,007.14497.

$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1.



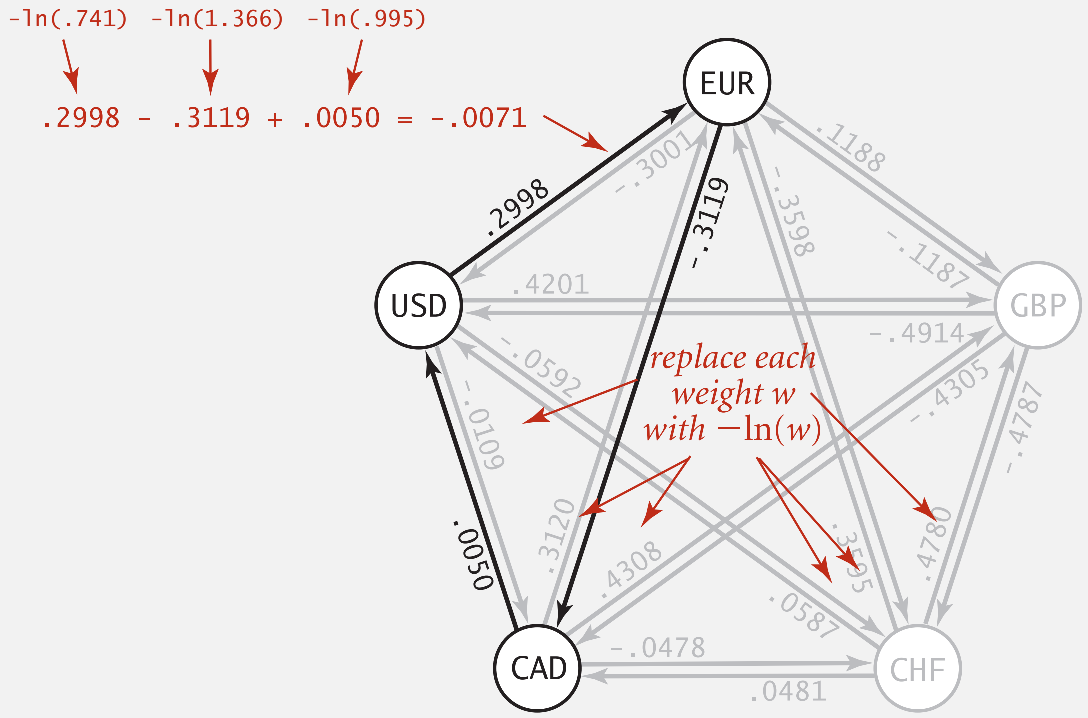
0.741 * 1.366 * .995 = 1.00714497

Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \to w$ be $-\ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0.
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.


Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.4 SHORTEST PATHS

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*