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

0.26 + 0.34 + 0.39 + 0.52 = 1.51
Google maps
Shortest path applications

- PERT/CPM.
- Map routing.
- **Seam carving.**
- Texture mapping.
- Robot navigation.
- 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.

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?

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

Cycles?

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

Simplifying assumption. Each vertex is reachable from $s$. 

which variant?
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.

```java
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 adjacent 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

adj

0

1

2

3

4

5

6

7

tinyEWD.txt

V

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
Edge-weighted digraph: adjacency-lists implementation in Java

Same as EdgeWeightedGraph except replace Graph with Digraph.

```java
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class <em>SP</em></td>
<td></td>
</tr>
<tr>
<td><code>SP(EdgeWeightedDigraph G, int s)</code></td>
<td>shortest paths from $s$ in graph $G$</td>
</tr>
<tr>
<td><code>double distTo(int v)</code></td>
<td>length of shortest path from $s$ to $v$</td>
</tr>
<tr>
<td><code>Iterable &lt;DirectedEdge&gt; pathTo(int v)</code></td>
<td>shortest path from $s$ to $v$</td>
</tr>
<tr>
<td><code>boolean hasPathTo(int v)</code></td>
<td>is there a path from $s$ to $v$?</td>
</tr>
</tbody>
</table>
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:
- $\text{distTo}[v]$ is length of shortest path from $s$ to $v$.
- $\text{edgeTo}[v]$ is last edge on shortest path from $s$ to $v$.

\[
\begin{array}{|c|c|c|}
\hline
\text{edgeTo[]} & \text{distTo[]} \\
\hline
0 & null & 0 \\
1 & 5\rightarrow1 & 0.32 & 1.05 \\
2 & 0\rightarrow2 & 0.26 & 0.26 \\
3 & 7\rightarrow3 & 0.37 & 0.97 \\
4 & 0\rightarrow4 & 0.38 & 0.38 \\
5 & 4\rightarrow5 & 0.35 & 0.73 \\
6 & 3\rightarrow6 & 0.52 & 1.49 \\
7 & 2\rightarrow7 & 0.34 & 0.60 \\
\hline
\end{array}
\]

shortest-paths tree from 0

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:

- \( \text{distTo}[v] \) is length of shortest path from \( s \) to \( v \).
- \( \text{edgeTo}[v] \) is last edge on shortest path from \( s \) to \( v \).

```java
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 $e = v \rightarrow w$.

- $\text{distTo}[v]$ is length of shortest known path from $s$ to $v$.
- $\text{distTo}[w]$ is length of shortest known path from $s$ to $w$.
- $\text{edgeTo}[w]$ is last edge on shortest known path from $s$ to $w$.
- If $e = v \rightarrow w$ gives shorter path to $w$ through $v$, update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

$v \rightarrow w$ successfully relaxes
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 $e = v \rightarrow w$ gives shorter path to $w$ through $v$, update both distTo[w] and edgeTo[w].

```java
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;
    }
}
```
Let $e = v \rightarrow w$ be an edge with weight 17.0. Suppose that $\text{distTo}[v] = \infty$ and $\text{distTo}[w] = 15.0$. Which is the value of $\text{distTo}[w]$ after calling $\text{relax}(e)$?

A. The program will throw a `java.lang.RuntimeException`.
B. $+\infty$
C. 17.0
D. 15.0
E. I don't know.
Shortest-paths optimality conditions

**Proposition.** Let $G$ be an edge-weighted digraph.
Then $\text{distTo}[]$ are the shortest path distances from $s$ iff:

- $\text{distTo}[s] = 0$.
- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e$.weight().

**Pf.** $\Leftarrow$ [ necessary ]

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e$.weight() for some edge $e = v \rightarrow w$.
- Then, $e$ gives a path from $s$ to $w$ (through $v$) of length less than $\text{distTo}[w]$.
Shortest-paths optimality conditions

**Proposition.** Let $G$ be an edge-weighted digraph.

Then $\text{distTo}[\cdot]$ are the shortest path distances from $s$ iff:

- $\text{distTo}[s] = 0$.
- For each vertex $v$, $\text{distTo}[v]$ is the length of some path from $s$ to $v$.
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

**Pf.** $\Rightarrow$ [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, $\text{distTo}[v_1] \leq \text{distTo}[v_0] + e_1.\text{weight}()$
  \[ \text{distTo}[v_2] \leq \text{distTo}[v_1] + e_2.\text{weight}() \]
  \[ \ldots \]
  \[ \text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}() \]

- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:
  \[ \text{distTo}[w] = \text{distTo}[v_k] \leq e_1.\text{weight}() + e_2.\text{weight}() + \ldots + e_k.\text{weight}() \]

- Thus, $\text{distTo}[w]$ is the weight of shortest path to $w$. ■
**Generic shortest-paths algorithm**

**Generic algorithm (to compute a SPT from s)**

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

Repeat until optimality conditions are satisfied:
  - Relax an edge.

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

**Pf sketch.**
- distTo[v] is always the length of a simple path from s to v.
- Each successful relaxation decreases distTo[v] for some v.
- distTo[v] can decrease at most a finite number of times. ■
Generic shortest-paths algorithm

Generic algorithm (to compute a SPT from s)

Initialize \( \text{distTo}[s] = 0 \) and \( \text{distTo}[v] = \infty \) for all other vertices.

Repeat until optimality conditions are satisfied:
- Relax an 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).
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.”

“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.”

http://catpad.net/michael/apl
“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

-- Edsger Dijkstra
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 adjacent from that vertex.

An edge-weighted digraph
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 adjacent from that vertex.

shortest-paths tree from vertex s
Dijkstra's algorithm visualization
Dijkstra's algorithm: correctness proof 1

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

**Pf.**
- Each edge $e = v \rightarrow w$ is relaxed exactly once (when vertex $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$ cannot increase
  - $\text{distTo}[v]$ will not change

Thus, upon termination, shortest-paths optimality conditions hold. ■
Dijkstra's algorithm: Java implementation

```java
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
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]);
    }
}
Shortest paths: quiz 2

What is the order of growth of the running time of Dijkstra's algorithm when using a binary heap for the priority queue?

A. $V + E$
B. $V \log E$
C. $E \log V$
D. $E \log E$
E. *I don't know.*
Computing a spanning tree in a graph

Dijkstra's algorithm seem familiar?

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

**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 this family of algorithms.
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 adjacent 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 adjacent from that vertex.

<table>
<thead>
<tr>
<th>v</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

shortest-paths tree from vertex s
What is the order of growth of the running time of the topological sort algorithm for computing shortest paths in an edge-weighted DAG?

A. $V$
B. $E$
C. $V + E$
D. $V \log E$
E. I don't know.
Shortest paths in edge-weighted DAGs

```java
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);
    }
}
```
Content-aware resizing

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

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, 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).
**Shortest Path Variants in a Digraph**

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

![Graph with vertex weights](image1)

**Q2.** How to model multiple sources and sinks?

![Graph with multiple sources and sinks](image2)
**Challenge.** Given an edge-weighted DAG, find the longest path from $s$ to any other vertex

**Warning.** Problem in general digraphs is NP-COMPLETE.

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

```
longest path from 5 to 0
(0.32 + 0.29 + 0.52 + 0.93 + 0.38 = 2.44)
```
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.

\[
\begin{align*}
\text{longest paths input} & \quad \text{shortest paths input} \\
5->4 & 0.35 \quad 5->4 & -0.35 \\
4->7 & 0.37 \quad 4->7 & -0.37 \\
5->7 & 0.28 \quad 5->7 & -0.28 \\
5->1 & 0.32 \quad 5->1 & -0.32 \\
4->0 & 0.38 \quad 4->0 & -0.38 \\
0->2 & 0.26 \quad 0->2 & -0.26 \\
3->7 & 0.39 \quad 3->7 & -0.39 \\
1->3 & 0.29 \quad 1->3 & -0.29 \\
7->2 & 0.34 \quad 7->2 & -0.34 \\
6->2 & 0.40 \quad 6->2 & -0.40 \\
3->6 & 0.52 \quad 3->6 & -0.52 \\
6->0 & 0.58 \quad 6->0 & -0.58 \\
6->4 & 0.93 \quad 6->4 & -0.93 \\
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>job</th>
<th>duration</th>
<th>must complete before</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41.0</td>
<td>1 7 9</td>
</tr>
<tr>
<td>1</td>
<td>51.0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>50.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36.0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>38.0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21.0</td>
<td>3 8</td>
</tr>
<tr>
<td>7</td>
<td>32.0</td>
<td>3 8</td>
</tr>
<tr>
<td>8</td>
<td>32.0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>29.0</td>
<td>4 6</td>
</tr>
</tbody>
</table>
**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).

```
<table>
<thead>
<tr>
<th>job</th>
<th>duration</th>
<th>must complete before</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41.0</td>
<td>1 7 9</td>
</tr>
<tr>
<td>1</td>
<td>51.0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>50.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36.0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>38.0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21.0</td>
<td>3 8</td>
</tr>
<tr>
<td>7</td>
<td>32.0</td>
<td>3 8</td>
</tr>
<tr>
<td>8</td>
<td>32.0</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>29.0</td>
<td>4 6</td>
</tr>
</tbody>
</table>
```
Critical path method

**CPM.** Use longest path from the source to schedule each job.
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 the vertices in the order 0, 3, 2, 1
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 8 to each edge weight changes the shortest path from 0→1→2→3 to 0→3.

**Conclusion.** Need a different algorithm.
Negative cycles

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

**Proposition.** A SPT exists iff no negative cycles.
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);
Bellman–Ford algorithm demo

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

an edge-weighted digraph
Bellman–Ford algorithm demo

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

shortest-paths tree from vertex $s$

<table>
<thead>
<tr>
<th>$v$</th>
<th>distTo[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5→2</td>
</tr>
<tr>
<td>3</td>
<td>17.0</td>
<td>2→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4→5</td>
</tr>
<tr>
<td>6</td>
<td>25.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
Bellman–Ford algorithm: visualization

passes
4

7

10

13

SPT
Bellman–Ford algorithm: analysis

Bellman–Ford algorithm

Initialize \( \text{distTo}[s] = 0 \) and \( \text{distTo}[v] = \infty \) for all other vertices.

Repeat \( V \) times:
- Relax each edge.

**Proposition.** Bellman–Ford 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 to each vertex \( v \) for which the shortest path from \( s \) to \( v \) contains \( i \) edges (or fewer).
Bellman–Ford algorithm: practical improvement

**Observation.** If \( \text{distTo}[v] \) does not change during pass \( i \), no need to relax any edge adjacent from \( v \) in pass \( i+1 \).

**FIFO implementation.** Maintain queue of vertices whose \( \text{distTo}[] \) 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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>restriction</th>
<th>typical case</th>
<th>worst case</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>topological sort</td>
<td>no directed cycles</td>
<td>$E + V$</td>
<td>$E + V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Dijkstra (binary heap)</td>
<td>no negative weights</td>
<td>$E \log V$</td>
<td>$E \log V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Bellman–Ford</td>
<td>no negative cycles</td>
<td>$E V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Bellman–Ford (queue–based)</td>
<td></td>
<td>$E + V$</td>
<td>$E V$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

**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 methods to the API for SP.

<table>
<thead>
<tr>
<th>boolean</th>
<th>hasNegativeCycle()</th>
<th>is there a negative cycle?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterable</td>
<td>&lt;DirectedEdge&gt;</td>
<td>negative cycle reachable from s</td>
</tr>
</tbody>
</table>

**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

```
5->4->7->5...
```

graph

```
5
  `-> 7
|    |    |
|    |    v
4    6
    `-> 3
      |    v
      |    2
      |    `-> 0
      |      `-> 1
      |        v
      |        7
      |        `-> 4
```

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.

![Graph](image)

**Proposition.** If Bellman–Ford updates any vertex $v$ in pass $V$, there exists a negative cycle (and can trace `edgeTo[v]` entries back to find one).

**In practice.** Check for negative cycles more frequently.
Negative cycle application: arbitrage detection

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

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>GBP</th>
<th>CHF</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>1</td>
<td>0.741</td>
<td>0.657</td>
<td>1.061</td>
<td>1.011</td>
</tr>
<tr>
<td>EUR</td>
<td>1.350</td>
<td>1</td>
<td>0.888</td>
<td>1.433</td>
<td>1.366</td>
</tr>
<tr>
<td>GBP</td>
<td>1.521</td>
<td>1.126</td>
<td>1</td>
<td>1.614</td>
<td>1.538</td>
</tr>
<tr>
<td>CHF</td>
<td>0.943</td>
<td>0.698</td>
<td>0.620</td>
<td>1</td>
<td>0.953</td>
</tr>
<tr>
<td>CAD</td>
<td>0.995</td>
<td>0.732</td>
<td>0.650</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497\]
Arbitrage

There's No Such Thing As a Free Lunch
Milton Friedman
Essays on Public Policy
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 $> 1$.

Challenge. Express as a negative cycle detection problem.
Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logarithms.

- Set weight of edge $v \rightarrow w$ to $-\ln$ (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

Nonnegative weights.
- Arises in many application.
- Dijkstra's algorithm is nearly linear-time.

Acyclic edge-weighted digraphs.
- Arise in some applications.
- Topological sort algorithm is linear time.
- Edge weights can be negative.

Negative weights and negative cycles.
- Arise in some applications.
- Bellman–Ford is quadratic in worst case.
- 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.