4.4 Shortest Paths

- edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights
**Shortest paths in a weighted digraph**

Given an edge-weighted digraph, find the shortest (directed) path from \( s \) to \( t \).
Google maps
Continental U.S. routes (August 2010)

http://www.continental.com/web/en-US/content/travel/routes
Shortest path applications

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

Shortest path variants

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

Restrictions on edge weights?
• Nonnegative weights.
• Arbitrary weights.
• Euclidean weights.

Cycles?
• No directed cycles.
• No "negative cycles."

Simplifying assumption. There exists a shortest path from $s$ to each vertex $v$. 
edge-weighted digraph API
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights
Weighted directed edge API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class DirectedEdge</td>
<td></td>
</tr>
<tr>
<td>DirectedEdge(int v, int w, double weight)</td>
<td>weighted edge v→w</td>
</tr>
<tr>
<td>int from()</td>
<td>vertex v</td>
</tr>
<tr>
<td>int to()</td>
<td>vertex w</td>
</tr>
<tr>
<td>double weight()</td>
<td>weight of this edge</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation</td>
</tr>
</tbody>
</table>

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 (int) 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

adj

0

1

2

3

4

5

6

7

Bag objects

reference to a DirectedEdge object

0 2 0.26 → 0 4 0.38

1 3 0.29

2 7 0.34

3 6 0.52

4 7 0.37 → 4 5 0.35

5 1 0.32

6 4 0.93

7 3 0.39

0 4 0.38

0 2 0.26

1 3 0.29

2 7 0.34

3 6 0.52

4 7 0.37

5 1 0.32

6 2 0.40

6 0 0.58

6 4 0.93

7 3 0.39

7 5 0.28
Same as `EdgeWeightedGraph` except replace `Graph` with `Digraph`.

```java
public class EdgeWeightedDigraph
{
   private final int V;
   private final Bag<Edge>[] 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` only to `v`'s adjacency list.
**Single-source shortest paths API**

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

```java
public class SP

SP(EdgeWeightedDigraph G, int s)  // shortest paths from s in graph G
   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 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.

```java
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?
```

```bash
% 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
```
edge-weighted digraph API
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 $v$. 

<table>
<thead>
<tr>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>5→1</td>
<td>0.32</td>
</tr>
<tr>
<td>0→2</td>
<td>0.26</td>
</tr>
<tr>
<td>7→3</td>
<td>0.37</td>
</tr>
<tr>
<td>0→4</td>
<td>0.38</td>
</tr>
<tr>
<td>4→5</td>
<td>0.35</td>
</tr>
<tr>
<td>3→6</td>
<td>0.52</td>
</tr>
<tr>
<td>2→7</td>
<td>0.34</td>
</tr>
</tbody>
</table>

shortest-paths tree from 0
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 $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

$v \rightarrow w$ successfully relaxes
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 $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

```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;
    }
}
```
Shortest-paths optimality conditions

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

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

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

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

![Diagram](image_url)
Shortest-paths optimality conditions

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

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

**Pf.** $\Rightarrow$ [sufficient]

- Suppose that $s = v_0 \to v_1 \to v_2 \to \cdots \to v_k = w$ is a shortest path from $s$ to $w$.
- Then,
  
  
  \[
  \begin{align*}
  \text{distTo}[v_k] & \leq \text{distTo}[v_{k-1}] + e_k\.weight() \\
  \text{distTo}[v_{k-1}] & \leq \text{distTo}[v_{k-2}] + e_{k-1}.weight() \\
  & \vdots \\
  \text{distTo}[v_1] & \leq \text{distTo}[v_0] + e_1\.weight()
  \end{align*}
  \]

- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:
  
  \[
  \text{distTo}[w] = \text{distTo}[v_k] \leq e_k\.weight() + e_{k-1}.weight() + \cdots + e_1\.weight()
  \]

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

- **Generic algorithm (to compute SPT from s)**
  
  Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ 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, $\text{distTo}[v]$ is the length of a simple path from $s$ to $v$ (and $\text{edgeTo}[v]$ is last edge on path).
- Each successful relaxation decreases $\text{distTo}[v]$ for some $v$.
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times. ■
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).

---

**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.
edge-weighted digraph API
shortest-paths properties
Dijkstra's algorithm
edge-weighted DAGs
negative weights
“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.”
“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 pointing from that vertex.
Dijkstra's algorithm demo

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

```
shortest-paths tree from vertex $s$
```

<table>
<thead>
<tr>
<th>$v$</th>
<th>$\text{distTo}[]$</th>
<th>$\text{edgeTo}[]$</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>
Dijkstra’s algorithm visualization
Dijkstra’s algorithm visualization
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 $v$ 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

  $\text{distTo}[]$ values are monotone decreasing
  edge weights are nonnegative and we choose lowest $\text{distTo}[]$ value at each step

• 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]);
    }
}
Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: \( V \) insert, \( V \) delete-min, \( E \) decrease-key.

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>insert</th>
<th>delete-min</th>
<th>decrease-key</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>1</td>
<td>( V )</td>
<td>1</td>
<td>( V^2 )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( \log V )</td>
<td>( \log V )</td>
<td>( \log V )</td>
<td>( E \log V )</td>
</tr>
<tr>
<td>d-way heap (Johnson 1975)</td>
<td>( d \log_d V )</td>
<td>( d \log_d V )</td>
<td>( \log_d V )</td>
<td>( E \log \frac{E}{V} V )</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman-Tarjan 1984)</td>
<td>( 1 \dagger )</td>
<td>( \log V \dagger )</td>
<td>( 1 \dagger )</td>
<td>( E + V \log V )</td>
</tr>
</tbody>
</table>

\( \dagger \) amortized

Bottom line.

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

**Insight.** Four of our graph-search methods are the same algorithm!
- Maintain a set of explored vertices \( S \).
- Grow \( S \) by exploring edges with exactly one endpoint leaving \( S \).

**DFS.** Take edge from vertex which was discovered most recently.
**BFS.** Take edge from vertex which was discovered least recently.
**Prim.** Take edge of minimum weight.
**Dijkstra.** Take edge to vertex that is closest to \( S \).

**Challenge.** Express this insight in reusable Java code.
- edge-weighted digraph API
- 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!
Topological sort algorithm demo

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

an edge-weighted DAG
• Consider vertices in topologically order.
• Relax all edges pointing from vertex.

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

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

```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.
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 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 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).

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

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

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

<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).**
Critical path method

**CPM.** Use *longest path* from the source to schedule each job.
edge-weighted digraph API
shortest-paths properties
Dijkstra's algorithm
dge-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.

**Bad news.**  Need a different algorithm.
**Def.** A negative cycle is a directed cycle whose sum of edge weights is negative.

**Proposition.** A SPT exists iff no negative cycles.

Assuming all vertices reachable from s.
Bellman-Ford algorithm

Bellman–Ford algorithm

Initialize \( \text{distTo}[s] = 0 \) and \( \text{distTo}[v] = \infty \) 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 $s$
Bellman-Ford algorithm visualization

passes
4
7
10

13
SPT
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.
Observation. If \( \text{distTo}[v] \) 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 \( \text{distTo}[] \) changed.

Overall effect.
• The running time is still proportional to \( E \times V \) in worst case.
• But much faster than that in practice.

Bellman-Ford algorithm: practical improvement
public class BellmanFordSP
{
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private boolean[] onQ;
    private Queue<Integer> queue;

    public BellmanFordSPT(EdgeWeightedDigraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        onq    = new boolean[G.V()];
        queue  = new Queue<Integer>();

        for (int v = 0; v < V; v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        queue.enqueue(s);
        while (!queue.isEmpty())
        {
            int v = queue.dequeue();
            onQ[v] = false;
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    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 (!onQ[w])
            {
                queue.enqueue(w);
                onQ[w] = true;
            }
        }
    }
}
**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>no negative cycles</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 method to the API for \(\texttt{SP}\).

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

digraph

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-&gt;5</td>
<td>0.35</td>
</tr>
<tr>
<td>5-&gt;4</td>
<td>-0.66</td>
</tr>
<tr>
<td>4-&gt;7</td>
<td>0.37</td>
</tr>
<tr>
<td>5-&gt;7</td>
<td>0.28</td>
</tr>
<tr>
<td>7-&gt;5</td>
<td>0.28</td>
</tr>
<tr>
<td>5-&gt;1</td>
<td>0.32</td>
</tr>
<tr>
<td>0-&gt;4</td>
<td>0.38</td>
</tr>
<tr>
<td>0-&gt;2</td>
<td>0.26</td>
</tr>
<tr>
<td>7-&gt;3</td>
<td>0.39</td>
</tr>
<tr>
<td>1-&gt;3</td>
<td>0.29</td>
</tr>
<tr>
<td>2-&gt;7</td>
<td>0.34</td>
</tr>
<tr>
<td>6-&gt;2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-&gt;6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-&gt;0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-&gt;4</td>
<td>0.93</td>
</tr>
</tbody>
</table>

negative cycle \((-0.66 + 0.37 + 0.28)\)

5->4->7->5

shortest path from 0 to 6
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.

![Diagram showing a cycle](image)

**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?

<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><strong>USD</strong></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EUR</strong></td>
<td>1.350</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GBP</strong></td>
<td>1.521</td>
<td>1.126</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CHF</strong></td>
<td>0.943</td>
<td>0.698</td>
<td>0.620</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>CAD</strong></td>
<td>0.995</td>
<td>0.732</td>
<td>0.650</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[
1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497
\]
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.
Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ 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!

Negative cycle application: arbitrage detection
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.