Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
Today

- Shortest Paths
- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- Negative weights
Shortest Paths

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- Negative weights
Given an edge-weighted digraph, find the shortest (directed) path from $s$ to $t$. 

**edge-weighted digraph**

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

**shortest path from 0 to 6**

- $0 \rightarrow 2$ 0.26
- $2 \rightarrow 7$ 0.34
- $7 \rightarrow 3$ 0.39
- $3 \rightarrow 6$ 0.52
Google maps
Car navigation
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.** Shortest paths from $s$ to each vertex $v$ exist.
Shortest Paths

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- Negative weights
Weighted directed edge API

```java
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

```java
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
  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
```

```
E
[[0, 2.6, 0, 4.38],
 [1, 3.29],
 [2, 7.34],
 [3, 6.52],
 [4, 7.37, 4, 5.35],
 [5, 1.32, 5, 7.28, 5, 4.35],
 [6, 4.93, 6, 0.58, 6, 2.40],
 [7, 3.39, 7, 5.28]]
```

Bag objects

reference to a DirectedEdge object
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 $e = v \rightarrow w$ only to $v$'s adjacency list
**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 $v$?

```
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)  \hspace{1cm} \text{shortest paths from } s \text{ in graph } G

double distTo(int v) \hspace{1cm} \text{length of shortest path from } s \text{ to } v

Iterable <DirectedEdge> pathTo(int v) \hspace{1cm} \text{shortest path from } s \text{ to } v

boolean hasPathTo(int v) \hspace{1cm} \text{is there a path from } s \text{ 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
Shortest Paths

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- Negative weights
**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 $v$. 

![shortest-paths tree from 0](image)
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 \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \).

\[ v \rightarrow w \text{ successfully relaxes} \]
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;
    }
}
```
**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 $e = v \rightarrow w$, \( \text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight()} \).

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

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

**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, for $k \geq 1$,
  
  $\text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight()}$  
  $\text{distTo}[v_{k-1}] \leq \text{distTo}[v_{k-2}] + e_{k-1}.\text{weight()}$  
  ...  

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

- 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. ■
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.

**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).
**Shortest Paths**

- 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

- 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 incident from that vertex.

\[ \begin{align*}
0 \rightarrow 1 & \quad 5.0 \\
0 \rightarrow 4 & \quad 9.0 \\
0 \rightarrow 7 & \quad 8.0 \\
1 \rightarrow 2 & \quad 12.0 \\
1 \rightarrow 3 & \quad 15.0 \\
1 \rightarrow 7 & \quad 4.0 \\
2 \rightarrow 3 & \quad 3.0 \\
2 \rightarrow 6 & \quad 11.0 \\
3 \rightarrow 6 & \quad 9.0 \\
4 \rightarrow 5 & \quad 4.0 \\
4 \rightarrow 6 & \quad 20.0 \\
4 \rightarrow 7 & \quad 5.0 \\
5 \rightarrow 2 & \quad 1.0 \\
5 \rightarrow 6 & \quad 13.0 \\
7 \rightarrow 5 & \quad 6.0 \\
7 \rightarrow 2 & \quad 7.0
\end{align*} \]
Dijkstra's algorithm

- 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 incident from that vertex.

Choose source vertex 0

<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></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra's algorithm

- 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 incident from that vertex.

### Example

```
<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></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Graph

- Relax all edges incident from 0.
Dijkstra's algorithm

- 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.
Dijkstra's algorithm

- 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.
Dijkstra's algorithm

- 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 incident from that vertex.

\[
\begin{array}{ccc}
\text{v} & \text{distTo[]} & \text{edgeTo[]} \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & & \\
3 & & \\
4 & 9.0 & 0\rightarrow4 \\
5 & & \\
6 & & \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}
\]

choose vertex 1
Dijkstra's algorithm

- 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 incident from that vertex.

relax all edges incident from 1
Dijkstra's algorithm

- 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 incident from that vertex.

Relax all edges incident from 1
**Dijkstra's algorithm**

- 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.

<table>
<thead>
<tr>
<th>( v )</th>
<th><code>distTo[]</code></th>
<th><code>edgeTo[]</code></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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
Dijkstra's algorithm

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

Choose vertex 7

<table>
<thead>
<tr>
<th>( v )</th>
<th>\texttt{distTo[]}</th>
<th>\texttt{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\rightarrow1</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1\rightarrow2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1\rightarrow3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0\rightarrow4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0\rightarrow7</td>
</tr>
</tbody>
</table>
Dijkstra's algorithm

- 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 incident 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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

relax all edges incident from 7
Dijkstra's algorithm

• 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 incident from that vertex.

relax all edges incident from 7
Dijkstra's algorithm

- 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 incident from that vertex.
Dijkstra's algorithm

- 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 incident 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</td>
</tr>
<tr>
<td>2</td>
<td>15.0</td>
<td>7→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>14.0</td>
<td>7→5</td>
</tr>
<tr>
<td>6</td>
<td>14.0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
```

select vertex 4
Dijkstra's algorithm

- 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 incident 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>15.0</td>
<td>7→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>14.0</td>
<td>7→5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
```

relax all edges incident from 4
Dijkstra's algorithm

- 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.
Dijkstra's algorithm

- 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 incident from that vertex.

<table>
<thead>
<tr>
<th>Vertex</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>15.0</td>
<td>7→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→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>29.0</td>
<td>4→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
Dijkstra's algorithm

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

Select vertex 5
Dijkstra's algorithm

- 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 incident from that vertex.
Dijkstra's algorithm

- 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 incident from that vertex.

relax all edges incident from 5
Dijkstra's algorithm

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

\begin{tabular}{c|c|c}
\textbf{v} & \textbf{distTo[]} & \textbf{edgeTo[]} \\
\hline
0 & 0.0 & - \\
1 & 5.0 & 0$\rightarrow$1 \\
2 & 14.0 & 5$\rightarrow$2 \\
3 & 20.0 & 1$\rightarrow$3 \\
4 & 9.0 & 0$\rightarrow$4 \\
5 & 13.0 & 4$\rightarrow$5 \\
6 & 26.0 & 5$\rightarrow$6 \\
7 & 8.0 & 0$\rightarrow$7 \\
\end{tabular}
Dijkstra's algorithm

- 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 incident from that vertex.

select vertex 2
Dijkstra's algorithm

- 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 incident 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>20.0</td>
<td>1→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>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

relax all edges incident from 2
Dijkstra's algorithm

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

relax all edges incident from 2
Dijkstra's algorithm

- 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 incident from that vertex.
Dijkstra's algorithm

- 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 incident from that vertex.
• 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 incident from that vertex.
Dijkstra's algorithm

• 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 incident from that vertex.

relax all edges incident from 3
Dijkstra's algorithm

- 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 incident from that vertex.
Dijkstra's algorithm

- 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 incident from that vertex.

![Graph with vertices and edge distances]

select vertex 6
• 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 incident from that vertex.

Dijkstra's algorithm

relax all edges incident from 6
Dijkstra's algorithm

- 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 incident from that vertex.
Dijkstra's algorithm

- 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 incident from that vertex.
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 → $\text{distTo}[]$ values are monotone decreasing
  - $\text{distTo}[v]$ will not change → edge weights are nonnegative and we choose lowest $\text{distTo}[]$ value at each step

- Thus, upon termination, shortest-paths optimality conditions hold. □
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>$E \log_{E/V} V$</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.
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.
Shortest Paths

- 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

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

an edge-weighted DAG
Consider vertices in topological order.
Relax all edges incident from that vertex.

topological order: 0 1 4 7 5 2 3 6
Topological sort algorithm

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

Choose vertex 0

```
choose vertex 0
```

```
0 1 4 7 5 2 3 6
v  distTo[]  edgeTo[]
0  0.0        -
1
2
3
4
5
6
7
```
Topological sort algorithm

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

relax all edges incident from 0
Topological sort algorithm

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

![Diagram of a graph with vertices and edges labeled with distances and directions.]

### relax all edges incident from 0

<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>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

### Example

**Graph:**

```
0 1 4 7 5 2 3 6
```

<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>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

### Algorithm

1. Choose vertex 1.
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

```
0 1 4 7 5 2 3 6

v distTo[] edgeTo[]
0 0.0 -
1 5.0 0→1
2 17.0 1→2
3 20.0 1→3
4 9.0 0→4
5
6
7 8.0 ✔ 0→7
```

relax all edges incident from 1
• Consider vertices in topological order.
• Relax all edges incident from that vertex.

Topological sort algorithm
Topological sort algorithm

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

select vertex 4
(Dijkstra would have selected vertex 7)
Consider vertices in topological order.
Relax all edges incident from that vertex.
Topological sort algorithm

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

relax all edges incident from 4

\[
\begin{array}{c|c|c}
\text{v} & \text{distTo[]} & \text{edgeTo[]} \\
\hline
0 & 0.0 & - \\
1 & 5.0 & 0→1 \\
2 & 17.0 & 1→2 \\
3 & 20.0 & 1→3 \\
4 & 9.0 & 0→4 \\
5 & 13.0 & 4→5 \\
6 & 29.0 & 4→6 \\
7 & 8.0 & 0→7 \\
\end{array}
\]
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

Choose vertex 7
• Consider vertices in topological order.
• Relax all edges incident from that vertex.
• Consider vertices in topological order.
• Relax all edges incident from that vertex.

relax all edges incident from 7
Topological sort algorithm

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

```
0 1 4 7 5 2 3 6
v distTo[] edgeTo[]
0 0.0 -
1 5.0 0→1
2 15.0 7→2
3 20.0 1→3
4 9.0 0→4
5 13.0 4→5
6 29.0 4→6
7 8.0 0→7
```
Topological sort algorithm

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

Select vertex 5

<table>
<thead>
<tr>
<th>Vertex</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</td>
</tr>
<tr>
<td>2</td>
<td>15.0</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>29.0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[89\]
**Topological sort algorithm**

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

```
relax all edges incident from 5
```

```
<table>
<thead>
<tr>
<th>vertex</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</td>
</tr>
<tr>
<td>2</td>
<td>15.0</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>29.0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Topological sort algorithm

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

relax all edges incident from 5
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

```
<table>
<thead>
<tr>
<th>vertex</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</td>
</tr>
<tr>
<td>2</td>
<td>14.0</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13.0</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>26.0</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

select vertex 2
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident 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>20.0</td>
<td>1→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>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
```

relax all edges incident from 2
• Consider vertices in topological order.
• Relax all edges incident from that vertex.
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

select vertex 3
Topological sort algorithm

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

relax all edges incident from 3
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.
Topological sort algorithm

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

select vertex 6
Topological sort algorithm

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

relax all edges incident from 6
Topological sort algorithm

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

shortest-paths tree from vertex $s$
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
  - $\text{distTo}[v]$ will not change

• 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, ...
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.
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.
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.

Key point. Topological sort algorithm works even with negative edge 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>
**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).**
CPM. Use **longest path** from the source to schedule each job.

Parallel job scheduling solution

**Critical path**
SHORTEST PATHS

- Edge-weighted digraph API
- Shortest-paths properties
- Dijkstra's algorithm
- Edge-weighted DAGs
- Negative weights
Dijkstra. Doesn’t work with negative edge weights.

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

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

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.
Repeat V times:
  - Relax each edge.

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

Bellman-Ford algorithm demo
Bellman-Ford algorithm demo

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

```
<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></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
pass 0
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
```
Repeat $V$ times: relax all $E$ edges.

**Bellman-Ford algorithm demo**

**pass 0**

<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>
</tbody>
</table>
Bellman-Ford algorithm demo

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

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

<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>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

<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></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

\[
\begin{array}{cccc}
v & \text{distTo[]} & \text{edgeTo[]} \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow 1 \\
2 & & \\
3 & & \\
4 & 9.0 & 0\rightarrow 4 \\
5 & & \\
6 & & \\
7 & 8.0 & 0\rightarrow 7 \\
\end{array}
\]

pass 0

0\rightarrow 1 0\rightarrow 4 0\rightarrow 7 1\rightarrow 2 1\rightarrow 3 1\rightarrow 7 2\rightarrow 3 2\rightarrow 6 3\rightarrow 6 4\rightarrow 5 4\rightarrow 6 4\rightarrow 7 5\rightarrow 2 5\rightarrow 6 7\rightarrow 5 7\rightarrow 2
Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

<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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

Repeat \( V \) times: relax all \( E \) edges.

<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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1  0→4  0→7  1→2  1→3  1→7  2→3  2→6  3→6  4→5  4→6  4→7  5→2  5→6  7→5  7→2
Repeat $V$ times: relax all $E$ edges.

**Bellman-Ford algorithm demo**

<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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

**pass 0**

0→1  0→4  0→7  1→2  1→3  1→7  2→3  2→6  3→6  4→5  4→6  4→7  5→2  5→6  7→5  7→2
Bellman-Ford algorithm demo

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

---

<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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0→4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

---

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Repeat $V$ times: relax all $E$ edges.
Repeat $V$ times: relax all $E$ edges.

Bellman-Ford algorithm demo

$\begin{array}{cccc}
v & \text{distTo[]} & \text{edgeTo[]} \\
0 & 0.0 & - \\
1 & 5.0 & 0\rightarrow1 \\
2 & 17.0 & 1\rightarrow2 \\
3 & 20.0 & 1\rightarrow3 \\
4 & 9.0 & 0\rightarrow4 \\
5 & & \\
6 & & \\
7 & 8.0 & 0\rightarrow7 \\
\end{array}$

pass 0

$0\rightarrow1$ $0\rightarrow4$ $0\rightarrow7$ $1\rightarrow2$ $1\rightarrow3$ $1\rightarrow7$ $2\rightarrow3$ $2\rightarrow6$ $3\rightarrow6$ $4\rightarrow5$ $4\rightarrow6$ $4\rightarrow7$ $5\rightarrow2$ $5\rightarrow6$ $7\rightarrow5$ $7\rightarrow2$
Bellman-Ford algorithm demo

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

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

Pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

Repeat \( V \) times: relax all \( E \) edges.

```
<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\rightarrow 1</td>
</tr>
<tr>
<td>2</td>
<td>17.0</td>
<td>1\rightarrow 2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1\rightarrow 3</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0\rightarrow 4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>28.0</td>
<td>2\rightarrow 6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0\rightarrow 7</td>
</tr>
</tbody>
</table>
```

pass 0

0\rightarrow 1 0\rightarrow 4 0\rightarrow 7 1\rightarrow 2 1\rightarrow 3 1\rightarrow 7 2\rightarrow 3 2\rightarrow 6 3\rightarrow 6 4\rightarrow 5 4\rightarrow 6 4\rightarrow 7 5\rightarrow 2 5\rightarrow 6 7\rightarrow 5 7\rightarrow 2
Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat \( V \) times: relax all \( E \) edges.

\[
\begin{array}{ccc}
\text{v} & \text{distTo[]} & \text{edgeTo[]} \\
0 & 0.0 & - \\
1 & 5.0 & 0 \rightarrow 1 \\
2 & 17.0 & 1 \rightarrow 2 \\
3 & 20.0 & 1 \rightarrow 3 \\
4 & 9.0 & 0 \rightarrow 4 \\
5 & 13.0 & 4 \rightarrow 5 \\
6 & 28.0 & 2 \rightarrow 6 \\
7 & 8.0 & 0 \rightarrow 7 \\
\end{array}
\]
Bellman-Ford algorithm demo

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

![Graph with distances and edge labels]

<table>
<thead>
<tr>
<th>Vertex</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>17.0</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>1→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>28.0</td>
<td>2→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

142
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

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

```
<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>20.0</td>
<td>1→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>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
```

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

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

---

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

---

$v$ distTo[] edgeTo[]

<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>20.0</td>
<td>1→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>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>
Bellman-Ford algorithm demo

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

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

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

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

pass 1

$\begin{array}{cccc}
0 & 1 & 2 & 3 \\
\rightarrow & 0 & 4 & 0 & 7 \\
1 & 1 & 2 & 1 \\
\rightarrow & 1 & 3 & 1 & 7 \\
2 & 2 & 6 & 3 & 6 \\
\rightarrow & 2 & 6 & 4 & 5 \\
3 & 4 & 5 & 4 & 6 \\
\rightarrow & 4 & 5 & 4 & 7 \\
4 & 5 & 2 & 5 & 6 \\
\rightarrow & 5 & 6 & 7 & 5 \\
5 & 7 & 2 \\
\end{array}$
Bellman-Ford algorithm demo

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

pass 1

$0 \rightarrow 1 \ 0 \rightarrow 4 \ 0 \rightarrow 7 \ 1 \rightarrow 2 \ 1 \rightarrow 3 \ 1 \rightarrow 7 \ 2 \rightarrow 3 \ 2 \rightarrow 6 \ 3 \rightarrow 6 \ 4 \rightarrow 5 \ 4 \rightarrow 6 \ 4 \rightarrow 7 \ 5 \rightarrow 2 \ 5 \rightarrow 6 \ 7 \rightarrow 5 \ 7 \rightarrow 2$
Bellman-Ford algorithm demo

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

- **distTo[]**
- **edgeTo[]**

<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>20.0</td>
<td>1→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>26.0</td>
<td>5→6</td>
</tr>
<tr>
<td>7</td>
<td>8.0</td>
<td>0→7</td>
</tr>
</tbody>
</table>

**pass 1**

0→1  0→4  0→7  1→2  1→3  1→7  2→3  2→6  3→6  4→5  4→6  4→7  5→2  5→6  7→5  7→2
Repeat $V$ times: relax all $E$ edges.

**Pass 1**

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

2-3 successfully relaxed in pass 1, but not pass 0

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

pass 1

$0 \rightarrow 1$ $0 \rightarrow 4$ $0 \rightarrow 7$ $1 \rightarrow 2$ $1 \rightarrow 3$ $1 \rightarrow 7$ $2 \rightarrow 3$ $2 \rightarrow 6$ $3 \rightarrow 6$ $4 \rightarrow 5$ $4 \rightarrow 6$ $4 \rightarrow 7$ $5 \rightarrow 2$ $5 \rightarrow 6$ $7 \rightarrow 5$ $7 \rightarrow 2$
Bellman-Ford algorithm demo

Repeat \( V \) times: relax all \( E \) edges.

---

2-6 successfully relaxed in pass 0 and pass 1

---

**pass 1**

\[
\begin{align*}
0 & \rightarrow 1 \\
0 & \rightarrow 4 \\
0 & \rightarrow 7 \\
1 & \rightarrow 2 \\
1 & \rightarrow 3 \\
1 & \rightarrow 7 \\
2 & \rightarrow 3 \\
2 & \rightarrow 6 \\
3 & \rightarrow 6 \\
4 & \rightarrow 5 \\
4 & \rightarrow 6 \\
4 & \rightarrow 7 \\
5 & \rightarrow 2 \\
5 & \rightarrow 6 \\
7 & \rightarrow 5 \\
7 & \rightarrow 2 \\
\end{align*}
\]
**Bellman-Ford algorithm demo**

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

Repeat $V$ times: relax all $E$ edges.
Bellman-Ford algorithm demo

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

**pass 1**

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

\[\text{pass 1}\]

\[
\begin{align*}
0 & \rightarrow 1 \\
0 & \rightarrow 4 \\
0 & \rightarrow 7 \\
1 & \rightarrow 2 \\
1 & \rightarrow 3 \\
1 & \rightarrow 7 \\
2 & \rightarrow 3 \\
2 & \rightarrow 6 \\
3 & \rightarrow 6 \\
4 & \rightarrow 5 \\
4 & \rightarrow 6 \\
4 & \rightarrow 7 \\
5 & \rightarrow 2 \\
5 & \rightarrow 6 \\
6 & \rightarrow 7 \\
7 & \rightarrow 5 \\
7 & \rightarrow 2
\end{align*}
\]
Bellman-Ford algorithm demo

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

pass 1

\[ 0 \rightarrow 1 \quad 0 \rightarrow 4 \quad 0 \rightarrow 7 \quad 1 \rightarrow 2 \quad 1 \rightarrow 3 \quad 1 \rightarrow 7 \quad 2 \rightarrow 3 \quad 2 \rightarrow 6 \quad 3 \rightarrow 6 \quad 4 \rightarrow 5 \quad 4 \rightarrow 6 \quad 4 \rightarrow 7 \quad 5 \rightarrow 2 \quad 5 \rightarrow 6 \quad 7 \rightarrow 5 \quad 7 \rightarrow 2 \]
Bellman-Ford algorithm demo

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

![Graph and table showing the Bellman-Ford algorithm in progress.

Table:

<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>

Diagram:

- Pass 1:
  - $0→1$
  - $0→4$
  - $0→7$
  - $1→2$
  - $1→3$
  - $1→7$
  - $2→3$
  - $2→6$
  - $3→6$
  - $4→5$
  - $4→6$
  - $4→7$
  - $5→2$
  - $5→6$
  - $7→5$
  - $7→2$
Bellman-Ford algorithm demo

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

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

**pass 1**

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
Bellman-Ford algorithm demo

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

pass 2, 3, 4, … (no further changes)

\begin{verbatim}
0 → 1 0 → 4 0 → 7 1 → 2 1 → 3 1 → 7 2 → 3 2 → 6 3 → 6 4 → 5 4 → 6 4 → 7 5 → 2 5 → 6 6 → 7
\end{verbatim}
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.
Bellman-Ford algorithm: practical improvement

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 $\text{distTo}[\cdot]$ 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.
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 `SP`.

```java
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

digraph

5->4->7->5
negative cycle (-0.66 + 0.37 + 0.28)

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

shortest path from 0 to 6

5->4->7->5
Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating \texttt{distTo[\cdot]} and \texttt{edgeTo[\cdot]} entries of vertices in the cycle.

![Diagram of a graph with cycles](image)

**Proposition.** If any vertex \( v \) is updated in phase \( V \), there exists a negative cycle (and can trace back \texttt{edgeTo[v]} entries to find it).

**In practice.** Check for negative cycles more frequently.
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>1.35</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>0.62</td>
<td>1</td>
<td>0.953</td>
</tr>
<tr>
<td>CAD</td>
<td>0.995</td>
<td>0.732</td>
<td>0.65</td>
<td>1.049</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

Example:

$$0.741 \times 1.366 \times 0.995 = 1.00714497$$
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!
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.