4.4 shortest paths

APIs
shortest-paths properties
Dijkstra’s algorithm
dge-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

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. Shortest paths from \( s \) to each vertex \( v \) exist.

Weighted directed edge API

public class DirectedEdge

```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; }
    public String toString() { return v + "->" + w; }
}
```

Idiom for processing an edge \( e \): \( \text{int } v = e.\text{from}(), w = e.\text{to}(); \)

Weighted directed edge: implementation in Java

Similar to Edge for undirected graphs, but a bit simpler.

```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;
    public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }
    public int from() { return v; }
    public int to() { return w; }
    public int weight() { return weight; }
}
```
**Edge-weighted digraph API**

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

    public EdgeWeightedDirectedGraph(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];
    }
}
```

**Conventions.** Allow self-loops and parallel edges.

**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];
    }
}
```

**Single-source shortest paths API**

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

```java
public class SP
{
    public double distTo(int v)
    {
        // length of shortest path from s to v
        return 0; // default value
    }

    public Iterable<DirectedEdge> pathTo(int v)
    {
        // shortest path from s to v
        return null; // default value
    }

    public boolean hasPathTo(int v)
    {
        // is there a path from s to v?
        return false; // default value
    }
}
```

**Example**

```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));
    if (sp.hasPathTo(v))
    {
        for (DirectedEdge e : sp.pathTo(v))
            StdOut.printf(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 $v$?

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 $v$?
```

---

**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 $v$.

![](shortest-paths-tree.png)

```
% java SP tinyEWD.txt 0
0 to 0 (0.00): 0->0 0.32
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
```

---

**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 $v$.

```
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 \( \text{edgeTo}[w] \).

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

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

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

**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, \( \text{distTo}[v_0] \leq \text{distTo}[v_1] \leq \text{distTo}[v_1] + \text{e_2.weight()} \)
  \[ \vdots \]
- \( \text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + \text{e_k.weight()} \)
  \( e_i = i^{th} \) edge on shortest path from \( s \) to \( w \)
- Add inequalities; simplify; and substitute \( \text{distTo}[v_0] = \text{distTo}[s] = 0; \)
  \( \text{distTo}[w] = \text{distTo}[v_k] \leq \text{e_1.weight()} + \text{e_2.weight()} + \ldots + \text{e_k.weight()} \)
  \( \text{weight of shortest path from } s \text{ to } w \)
- Thus, \( \text{distTo}[w] \) is the weight of shortest path to \( w \). \( \blacksquare \)
Generic shortest-paths algorithm

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

1. Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.
2. Repeat until optimality conditions are satisfied:
   - Relax any edge.

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

**Proof sketch.**
- The entry $\text{distTo}[v]$ is always the length of a simple path from $s$ to $v$.
- 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).

---

**Shortest paths: quiz 1**

Let $e = v \rightarrow w$ be an edge with weight 17.0. Suppose that during the generic shortest paths algorithm, $\text{distTo}[v] = \infty$ and $\text{distTo}[w] = 15.0$. What will $\text{distTo}[w]$ be after calling $\text{relax}(e)$?

- **A.** The program will throw a java.lang.RuntimeException.
- **B.** 15.0
- **C.** 17.0
- **D.** $+ \infty$
- **E.** I don’t know.
“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.”

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.

**Dijkstra's 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>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
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 when relaxing \( v \)
    - if \( u \) has not yet been relaxed, then \( \text{distTo}[u] \geq \text{distTo}[v] \)

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;

        pq.insert(s, 0.0);
        while (!pq.isEmpty())
        {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

relax vertices in order of distance from \( s \)
**Dijkstra’s algorithm: Java implementation**

```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;
        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>unordered 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</td>
<td>( \log_d V )</td>
<td>( d \log_d V )</td>
<td>( \log_d V )</td>
<td>( E \log_{dV} V )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>1 †</td>
<td>( \log V ) †</td>
<td>1 †</td>
<td>( E + V \log V )</td>
</tr>
</tbody>
</table>

† amortized

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

---

**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.
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.

Shortest paths in edge-weighted DAGs

**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 vertex $v$ is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + \text{e.weight}()$.
- 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

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.

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.

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

Content-aware resizing

http://www.youtube.com/watch?v=viFCV2spKtg
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.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>x</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>v'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q1. How to model both vertex and edge weights?

Q2. How to model multiple sources and sinks?

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 weights.

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.

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.
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.

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

Conclusion. Need a different algorithm.

Negative cycles

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

Bellman-Ford algorithm

Bellman–Ford algorithm

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

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

Proposition. A SPT exists iff no negative cycles (reachable from s).
Bellman-Ford algorithm demo

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

an edge–weighted digraph

Bellman-Ford algorithm: visualization

Bellman-Ford algorithm demo

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

shortest-paths tree from vertex $s$

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. 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 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 \texttt{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 \texttt{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.

Finding a negative cycle

Negative cycle. Add two methods to the API for SP.

\begin{verbatim}
boolean hasNegativeCycle()    // is there a negative cycle?
Iterable<DirectedEdge> negativeCycle()  // negative cycle reachable from \(s\)
\end{verbatim}

Finding a negative cycle

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
digraph
s
1
2
3
4
5
6
7

negative cycle (-0.66 + 0.37 + 0.28)
5->4->7->5
\end{verbatim}

Finding a negative cycle

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

\begin{verbatim}
edgeTo[v]
\end{verbatim}

Proposition. If Bellman-Ford updates any vertex \(v\) in pass \(v\), there exists a negative cycle (and can trace \texttt{edgeTo[v]} entries back to find one).

In practice. Check for negative cycles more frequently.

Single source shortest-paths implementation: cost summary

\begin{verbatim}
<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>
\end{verbatim}

Remark 1. Directed cycles make the problem harder.
Remark 2. Negative weights make the problem harder.
Remark 3. Negative cycles makes the problem intractable.
Negative cycle application: arbitrage detection

**Problem.** Given a 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.

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

Negative cycle application: arbitrage detection

**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).

\[-\ln(0.741) - \ln(1.366) - \ln(0.995)\]
\[0.2998 - 0.3119 + 0.0050 = -0.0071\]

**Remark.** Fastest algorithm is extraordinarily valuable!

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

\[0.741 \times 1.366 \times 0.995 = 1.00714497\]

Challenge. Express as a negative cycle detection problem.

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.**