4.4 Shortest Paths

- Dijkstra's algorithm
- implementation
- acyclic networks
- negative weights

Reference: Algorithms in Java, 4th edition, Section 4.4
Google maps
Shortest paths in a weighted digraph

Given a weighted digraph \( G \), find the shortest directed path from \( s \) to \( t \).

![Diagram of a weighted digraph]

shortest path: \( s \rightarrow 6 \rightarrow 3 \rightarrow 5 \rightarrow t \)

cost: \( 14 + 18 + 2 + 16 = 50 \)
Shortest path versions

Which vertices?
- From one vertex to another.
- From one vertex to every other.
- Between all pairs of vertices.

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

Cycles?
Early history of shortest paths algorithms


Ford (1956). RAND, economics of transportation.


Shortest path applications

- Maps.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Subroutine in advanced algorithms.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

- Dijkstra's algorithm
- implementation
- acyclic networks
- negative weights
“The question of whether computers can think is like the question of whether submarines can swim.”

“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.”

“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.”
Single-source shortest-paths

**Input.** Weighted digraph $G$, source vertex $s$.

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

**Observation.** Use parent-link representation to store shortest path tree.

![Graph with source vertex $s$ and distances, edges, and markings]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>distTo[$v$]</td>
<td>0</td>
<td>15</td>
<td>9</td>
<td>32</td>
<td>45</td>
<td>34</td>
<td>14</td>
<td>50</td>
</tr>
<tr>
<td>edgeTo[$v$]</td>
<td>-</td>
<td>0→1</td>
<td>0→2</td>
<td>6→3</td>
<td>5→4</td>
<td>3→5</td>
<td>0→6</td>
<td>5→7</td>
</tr>
<tr>
<td>marked[$v$]</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Dijkstra's algorithm

Start with vertex s and greedily grow tree T
• find cheapest path ending in an edge e with exactly one endpoint in T
• add e to T
• continue until no edges leave T
Dijkstra's algorithm

Initialize $T$ to $s$, $distTo[s]$ to 0.
Repeat until $T$ contains all vertices reachable from $s$:
• find edge $e$ with $v$ in $T$ and $w$ not in $T$ that minimizes $distTo[v] + e.weight()$
Dijkstra's algorithm

Initialize $T$ to $s$, $\text{distTo}[s]$ to 0.
Repeat until $T$ contains all vertices reachable from $s$:
• find edge $e$ with $v$ in $T$ and $w$ not in $T$ that minimizes $\text{distTo}[v] + e\cdot\text{weight}()$
• set $\text{distTo}[w] = \text{distTo}[v] + e\cdot\text{weight}()$ and $\text{edgeTo}[w] = e$
• add $w$ to $T$
Dijkstra’s algorithm example

edge with v in T and w not in T

0→1 (.41)
0→5 (.29)
1→2 (.51)
1→4 (.32)
2→3 (.50)
3→0 (.45)
3→5 (.38)
4→2 (.32)
4→3 (.36)
5→1 (.29)
5→4 (.21)

edge in shortest path tree

4→2 (.82 = .50 + .32)
4→3 (.86 = .50 + .36)
1→2 (.92)
5→4 (.50)
1→4 (.73 = .41 + .32)
1→2 (.92 = .41 + .51)

0→1 (.41)
0→5 (.29)
1→2 (.51)
1→4 (.32)
2→3 (.50)
3→0 (.45)
3→5 (.38)
4→2 (.32)
4→3 (.36)
5→1 (.29)
5→4 (.21)
Invariant. For \( v \) in \( T \), \( \text{distTo}[v] \) is the length of the shortest path from \( s \) to \( v \).

**Pf.** (by induction on \(|T|\))
- Let \( w \) be next vertex added to \( T \).
- Let \( P^* \) be the \( s \rightarrow w \) path through \( v \).
- Consider any other \( s \rightarrow w \) path \( P \), and let \( x \) be first node on path outside \( T \).
- \( P \) is already as long as \( P^* \) as soon as it reaches \( x \) by greedy choice.
- Thus, \( \text{distTo}[w] \) is the length of the shortest path from \( s \) to \( w \).
Dijkstra's algorithm
implementation
acyclic networks
negative weights
Weighted digraph API

Nomenclature reset: “Weighted directed graph” = “Network”

public class DirectedEdge

DirectedEdge(int v, int w, double weight)  // create a weighted edge v→w
  int from()  // vertex v
  int to()  // vertex w
  double weight()  // the weight

public class Network

Network(int V)  // create an empty digraph with V vertices
Network(In in)  // create a digraph from input stream
void addEdge(DirectedEdge e)  // add a weighted edge from v to w
Iterable<DirectedEdge> adj(int v)  // return an iterator over edges leaving v
int V()  // return number of vertices
int E()  // return number of edges
Iterable<DirectedEdge> edges()  // return an iterator over all the network’s edges
public class Network
{
    private final int V;
    private final Bag<Edge>[] adj;

    public Network(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]; }

    public int V()
    { return V; }
}
Weighted directed edge: implementation in Java

```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;  }
}
```

similar to Edge for undirected weighted graphs, but simpler

from() and to() replace either() and other()
Shortest path data type

Design pattern.

• DijkstraSPT class is a Network client.
• Client query methods return distance and path iterator.

```java
public class DijkstraSPT {
    DijkstraSPT(Network G, int s) // shortest path 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
}
```

```java
In in = new In("network.txt");
Network G = new Network(in);
int s = 0, t = G.V() - 1;
DijkstraSPT spt = new DijkstraSPT(G, s);
StdOut.println("distance = " + spt.distTo(t));
for (DirectedEdge e : spt.pathTo(t))
    StdOut.println(e);
```
Dijkstra implementation challenge

Find edge \( e \) with \( v \) in \( S \) and \( w \) not in \( S \) that minimizes \( \text{distTo}[v] + e.\text{weight()} \).

How difficult?
• Intractable.
• \( O(E) \) time.
• \( O(V) \) time.
• \( O(\log E) \) time.
• \( O(\log^* E) \) time.
• Constant time.

try all edges

Dijkstra with a binary heap
Lazy vs. eager implementation

Issue:
- PQ contains edges from a vertex \( v \) in \( S \) to a vertex \( w \) not in \( S \).
- Adding \( w \) to the tree requires adding its incident edges to PQ.
- Some edges on the PQ become obsolete.

Obsolete edge:
- An edge that will never be added to the tree

Lazy approach
- Leave obsolete edges on PQ
- Check for obsolescence when removing

Eager approach
- Remove obsolete edges from PQ (need more sophisticated PQ)
- only need one edge per vertex
Lazy Dijkstra’s algorithm example

```
0→1  .41
0→5  .29
1→2  .51
1→4  .32
2→3  .50
3→0  .45
3→5  .38
4→2  .32
4→3  .36
5→1  .29
5→4  .21
```

```
0→5  (.29)
0→1  (.41)
5→4 (.50 = .29 + .21)
5→1 (.58 = .29 + .29)
1→4 (.73 = .41 + .32)
1→2 (.92 = .41 + .51)
```

```
1→4  (.73)
4→2  (.82 = .50 + .32)
4→3  (.86 = .50 + .36)
1→2  (.92)
```

```
4→3  (0.86)
1→2  (.92)
2→3  (1.32 = .82 + .50)
```
import java.util.Comparator;

public class LazyDijkstraSPT
{
    private boolean[] marked;
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private MinPQ<DirectedEdge> pq;

    private class ByDistanceFromSource implements Comparator<DirectedEdge>
    {
        public int compare(DirectedEdge e, DirectedEdge f)
        {
            double x = distTo[e.from()] + e.weight();
            double y = distTo[f.from()] + f.weight();
            if (x < y) return -1;
            else if (x > y) return +1;
            else return 0;
        }
    }

    public LazyDijkstra(Network G, int s)
    {
        marked = new boolean[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new MinPQ<DirectedEdge>(new ByDistanceFromSource());
        dijkstra(G, s);
    }
}

compare edges in pq by distTo[v] + e.weight()
Lazy implementation of Dijkstra's algorithm

```java
private void dijkstra(Network G, int s)
{
    visit(G, s);
    while (!pq.isEmpty())
    {
        DirectedEdge e = pq.delMin();
        int v = e.from(), w = e.to();
        if (marked[w]) continue;
        distTo[w] = e;
        distTo[w] = distTo[v] + e.weight();
        visit(G, w);
    }
}

private void visit(Network G, int v)
{
    marked[v] = true;
    for (DirectedEdge e : G.adj(w))
        if (!marked[e.to()]) pq.insert(e);
}
```
Dijkstra's algorithm running time

Proposition. Dijkstra's algorithm computes shortest paths in $O(E \log E)$ time.

Pf.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>

Improvements.

- Eagerly eliminate obsolete edges from PQ.
- Maintain on PQ at most one edge incident to each vertex $v$ not in $T$ $\Rightarrow$ at most $V$ edges on PQ.
- Use fancier priority queue: best in theory yields $O(E + V \log V)$. 
Remark. Dijkstra examines vertices in increasing distance from source.
Priority-first search

**Insight.** All 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.
Priority-first search: application example

Shortest s-t paths in Euclidean graphs (maps)
• Vertices are points in the plane.
• Edge weights are Euclidean distances.

A sublinear algorithm.
• Assume graph is already in memory.
• Start Dijkstra at s.
• Stop when you reach t.

Even better: exploit geometry
• For edge $v \rightarrow w$, use weight $d(v, w) + d(w, t) - d(v, t)$.
• Proof of correctness for Dijkstra still applies.
• In practice only $O(V^{1/2})$ vertices examined.
• Special case of A* algorithm

[Practical map-processing programs precompute many of the paths.]
Dijkstra's algorithm
implementation
acyclic networks
negative weights
Acyclic networks

Suppose that a network has no cycles.

Q. Is it easier to find shortest paths than in a general network?

A. Yes!

A. AND negative weights are no problem

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

Suppose that a network has no cycles.

Q. Is it easier to find shortest paths than in a general network?
A. Yes!
A. AND negative weights are no problem

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

Relax edge \( e \) from \( v \) to \( w \).
- \( \text{distTo}[v] \) is length of some path from \( s \) to \( v \).
- \( \text{distTo}[w] \) is length of some path from \( s \) to \( w \).
- If \( v \rightarrow w \) gives a shorter path to \( w \) through \( v \), update \( \text{distTo}[w] \) and \( \text{edgeTo}[w] \).

Initialization:
- \( \text{distTo}[s] = 0.0 \);
- all other \( \text{distTo}[] = \infty \)

```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 in acyclic networks

Algorithm:
- Consider vertices in topologically sorted order
- Relax all edges incident on vertex
Shortest paths in acyclic networks

Algorithm:
• Consider vertices in topologically sorted order
• Relax all edges incident on vertex

Proposition. Shortest path to each vertex is known before its edges are relaxed

Proof (strong induction)
• let v→w be the last edge on the shortest path from s to w.  
• v appears before w in the topological sort  
  - shortest path to v is known before its edges are relaxed  
  - v’s edges are relaxed before w’s edges are relaxed, including v→w  
• therefore, shortest path to w is known before w’s edges are relaxed.
Shortest paths in acyclic networks

```java
public class AcyclicNetworkSPT {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    public AcyclicNetworkSPT(Network G, int s) {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        for (int v = 0; v < G.V(); v++)
            distanceTo[v] = Double.POSITIVE_INFINITY;
        distanceTo[s] = 0.0;
        NetworkSort sort = new NetworkSort(G);
        for (int v : sort.topological())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}
```

Longest paths in acyclic networks

Algorithm:
• Negate all weights
• Find shortest path
• Negate weights in result

Note: Best known algorithm for general networks is exponential!
**Longest paths in acyclic networks: application**

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

**Ex:**

<table>
<thead>
<tr>
<th>job</th>
<th>duration</th>
<th>count</th>
<th>successors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41.0</td>
<td>3</td>
<td>1 7 9</td>
</tr>
<tr>
<td>1</td>
<td>51.0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>50.0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36.0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>38.0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>45.0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21.0</td>
<td>2</td>
<td>3 8</td>
</tr>
<tr>
<td>7</td>
<td>32.0</td>
<td>2</td>
<td>3 8</td>
</tr>
<tr>
<td>8</td>
<td>32.0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>29.0</td>
<td>2</td>
<td>4 6</td>
</tr>
</tbody>
</table>

**Solution:**

```
0  9  6  8  2
1  3
7
5  4
41  91  173
```


**Critical path method**

**CPM.** To solve a job-scheduling problem, create a network

- source, sink
- two vertices (begin and end) for each job
- three edges for each job
  - begin to end (weighted by duration)
  - source to begin
  - end to sink

**Critical path method:** Use longest path from the source to schedule each job

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

**CPM.** To solve a job-scheduling problem, create a network

- source, sink
- two vertices (begin and end) for each job
- three edges for each job
  - begin to end (weighted by duration)
  - source to begin
  - end to sink

**Critical path method:** Use longest path from the source to schedule each job
Critical path method

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

```
Critical path duration
```

```
0 9 6 8 2
0 7 1
0 5 4
0 2
0 3
0 4
```
Add **deadlines** to the job-scheduling problem.

Ex. “Job 2 must start no later than 70 time units after job 7.”
Or, “Job 7 must start no earlier than 70 times units before job 2”

Need to solve longest paths problem in general networks (cycles, neg weights).
Possibility of infeasible problem (negative cycles)
› Dijkstra's algorithm
› implementation
› negative weights
Shortest paths with negative weights: failed attempts

**Dijkstra.** Doesn’t work with negative edge weights.

![Graph](image)

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 also doesn’t work.

![Graph](image)

Adding 9 to each edge changes the shortest path because it adds 9 to each edge; wrong thing to do for paths with many edges.

**Bad news.** Need a different algorithm.
Negative cycles

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

**Observations.** If negative cycle $C$ is on a path from $s$ to $t$, then shortest path can be made arbitrarily negative by spinning around cycle.

**Worse news.** Need a different problem.
Shortest paths with negative weights

**Problem 1.** Does a given digraph contain a negative cycle?

**Problem 2.** Find the shortest simple path from $s$ to $t$.

*Bad news.* Problem 2 is intractable.

*Good news.* Can solve problem 1 in $O(VE)$ steps; if no negative cycles, can solve problem 2 with same algorithm!
Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

- Initialize $\text{distTo}[v] = \infty$, $\text{distTo}[s] = 0$.
- Repeat $v$ times: relax each edge $e$.

```java
for (int i = 1; i <= G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v)) relax(e);
```

phase $i$
Dynamic programming algorithm trace

relaxed edges that update distTo[]

0→1 (.41 = 0 + .41)
0→5 (.50 = 0 + .50)
1→2 (.73 = .41 + .32)
1→4 (.73 = .41 + .32)
5→4 (.50 = .29 + .21)

2→3 (1.33 = .83 + .50)
4→3 (.86 = .50 + .36)
4→2 (.82 = .50 + .32)

can stop early since no entries in distTo[] updated
Dynamic programming algorithm: analysis

Running time. Proportional to $E \times V$.

Invariant. At end of phase $i$, $\text{distTo}[v] \leq$ length of any path from $s$ to $v$ using at most $i$ edges.

Proposition. If there are no negative cycles, upon termination $\text{distTo}[v]$ is the length of the shortest path from $s$ to $v$. and $\text{edgeTo}[]$ gives the shortest paths
Bellman-Ford-Moore algorithm

**Observation.** If $\text{distTo}[v]$ doesn't change during phase $i$, no need to relax any edge leaving $v$ in phase $i+1$.

**FIFO implementation.** Maintain *queue* of vertices whose distance changed.

\[\uparrow\]
be careful to keep at most one copy of each vertex on queue

**Running time.**
- Proportional to $EV$ in worst case.
- Much faster than that in practice.
Bellman-Ford-Moore algorithm

```
public class BellmanFordSPT {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private int phase;
    private int[] beenTo;
    private Queue<Integer> q = new Queue<Integer>();
    private Queue<Integer> relaxed;

    public BellmanFordSPT(Network G, int s) {
        distTo = new double[V];
        edgeTo = new DirectedEdge[V];
        beenTo = new int[V];

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

        q.enqueue(s);
        distanceTo[s] = 0.0;
        for (phase = 1; phase <= V; phase++)
            relaxed = new Queue<Integer>();

            for (int v : q)
                for (DirectedEdge e : G.adj(v))
                    relax(e);

            q = relaxed;
            if (q.isEmpty()) break;
    }

    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 (beenTo[w] < phase)
                relaxed.enqueue(w);
            beenTo[w] = phase;
        }
    }

    public void maintainQueue() {
        // Maintain queue of vertices whose distance changes.
    }

    public void relaxEdges() {
        // Relax all edges incident on all vertices in the queue.
    }
}
```
**Single source shortest paths implementation: cost summary**

<table>
<thead>
<tr>
<th>no cycles</th>
<th>topological sort + relax</th>
<th>$E$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonnegative costs</td>
<td>Dijkstra (binary heap)</td>
<td>$E \log E$</td>
<td>$E$</td>
</tr>
<tr>
<td>no negative cycles</td>
<td>dynamic programming</td>
<td>$E V$</td>
<td>$E V$</td>
</tr>
<tr>
<td></td>
<td>Bellman-Ford</td>
<td>$E V$</td>
<td>$E$</td>
</tr>
</tbody>
</table>

**Remark 1.** Cycles make the problem harder.

**Remark 2.** Negative weights make the problem harder.

**Remark 3.** Negative cycles makes the problem intractable.
Problem. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

- 1 oz. gold \( \Rightarrow \) $327.25.
- 1 oz. gold \( \Rightarrow \) £208.10 \( \Rightarrow \) $327.00.
- 1 oz. gold \( \Rightarrow \) 455.2 Francs \( \Rightarrow \) 304.39 Euros \( \Rightarrow \) $327.28.

\[
\begin{align*}
\text{UK pound} & : 1.0000 & 0.6853 & 0.005290 & 0.4569 & 0.6368 & 208.100 \\
\text{Euro} & : 1.45999 & 1.0000 & 0.007721 & 0.6677 & 0.9303 & 304.028 \\
\text{Japanese Yen} & : 189.50 & 129.520 & 1.0000 & 85.4694 & 120.400 & 39346.7 \\
\text{Swiss Franc} & : 2.1904 & 1.4978 & 0.01574 & 1.0000 & 1.3941 & 455.200 \\
\text{US dollar} & : 1.5714 & 1.0752 & 0.008309 & 0.7182 & 1.0000 & 327.250 \\
\text{Gold (oz.)} & : 0.004816 & 0.003295 & 0.000255 & 0.002201 & 0.003065 & 1.0000
\end{align*}
\]
Currency conversion

Graph formulation.
• Vertex = currency.
• Edge = transaction, with weight equal to exchange rate.
• Find path that maximizes product of weights.

Challenge. Express as a shortest path problem.


Currency conversion

Reduce to shortest path problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $- \log$ (exchange rate from currency $v$ to $w$).
- Multiplication turns to addition.
- Shortest path with given weights corresponds to best exchange sequence.

**Challenge.** Solve shortest path problem with negative weights.
Shortest paths application: arbitrage

Is there an arbitrage opportunity in currency graph?
- Ex: $1 \Rightarrow 1.3941$ Francs $\Rightarrow 0.9308$ Euros $\Rightarrow $1.00084.
- Is there a negative cost cycle?

Remark. Fastest algorithm is valuable!
Negative cycle detection

If there is a negative cycle reachable from \( s \).
Bellman-Ford-Moore gets stuck in loop, updating vertices in cycle.

![Graph showing negative cycle](image)

**Proposition.** If any vertex \( v \) is updated in phase \( v \), there exists a negative cycle, and we can trace back \( \text{edgeTo}[v] \) to find it.
**Goal.** Identify a negative cycle (reachable from any vertex).

**Solution.** Initialize Bellman-Ford by setting $\text{distTo}[v] = 0$ for all vertices $v$ and putting all vertices on the queue.
Shortest paths summary

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

Acyclic networks.
- Arise in applications.
- Faster than Dijkstra’s algorithm.
- Negative weights are no problem.

Negative weights.
- Arise in applications.
- If negative cycles, shortest simple-paths problem is intractable (!)
- If no negative cycles, solvable via classic algorithms.

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