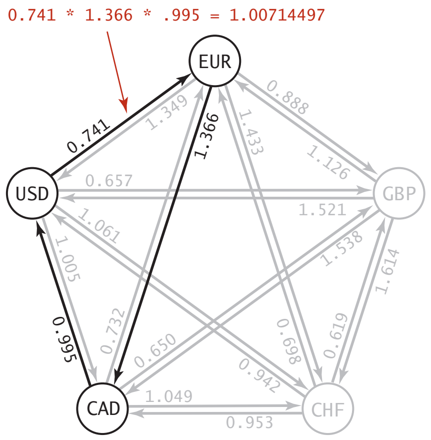
Arbitrage. Consider a market for financial transactions that is based on trading commodities. You can find a familiar example in tables that show conversion rates among currencies, such as the one in our sample file `rates.txt` shown here. The first line in the file is the number *V* of currencies; then the file has one line per currency, giving its name followed by the conversion rates to the other currencies. For brevity, this example includes just five of the hundreds of currencies that are traded on modern markets: U.S. dollars (`USD`), Euros (`EUR`), British pounds (`GBP`), Swiss francs (`CHF`), and Canadian dollars (`CAD`). The `t`th number on line `s` represents a conversion rate: the number of units of the currency named on row `s` that is needed to buy 1 unit of the currency named on row `t`. For example, our table says that 1,000 U.S. dollars will buy 741 euros. This table is equivalent to a *complete edge-weighted digraph* with a vertex corresponding to each currency and an edge corresponding to each conversion rate. An edge `s->t` with weight `x` corresponds to a conversion from `s` to `t` at exchange rate `x`. Paths in the digraph specify multistep conversions. For example, combining the conversion just mentioned with an edge `t->u` with weight `y` gives a path `s->t->u` that represents a way to convert 1 unit of currency `s` into `xy` units of currency `u`. For example, we might buy 1,012.206 = 741×1.366 Canadian dollars with our euros. Note that this gives a better rate than directly converting from U.S. dollars to Canadian dollars. You might expect `xy` to be equal to the weight of `s->u` in all such cases, but such tables represent a complex financial system where such consistency cannot be guaranteed. Thus, finding the path from `s` to `u` such that the product of the weights is maximal is certainly of interest. Even more interesting is a case where the product of the edge weights is *smaller* than the weight of the edge from the last vertex back to the first. In our example, suppose that the weight of `u->s` is `z` and `xyz > 1`. Then cycle `s->t->u->s` gives a way to convert 1 unit of currency `s` into more than 1 unit (`xyz`) of currency `s`. In other words, we can make a `100(xyz - 1)` percent profit by converting from `s` to `t` to `u` back to `s`. For example, if we convert our 1,012.206 Canadian dollars back to US dollars, we get 1,012.206*.995 = 1,007.14497 dollars, a 7.14497-dollar profit. That might not seem like

```
% more rates.txt
5
USD  1      0.741  0.657  1.061  1.005
EUR  1.349  1      0.888  1.433  1.366
GBP  1.521  1.126  1      1.614  1.538
CHF  0.942  0.698  0.619  1      0.953
CAD  0.995  0.732  0.650  1.049  1
```



An arbitrage opportunity

Arbitrage in currency exchange

```
public class Arbitrage
{
   public static void main(String[] args)
   {
      int V = StdIn.readInt();
      String[] name = new String[V];
      EdgeWeightedDigraph G = new EdgeWeightedDigraph(V);
      for (int v = 0; v < V; v++)
      {
         name[v] = StdIn.readString();
         for (int w = 0; w < V; w++)
         {
            double rate = StdIn.readDouble();
            DirectedEdge e = new DirectedEdge(v, w, -Math.log(rate));
            G.addEdge(e);
         }
      }

      BellmanFordSP spt = new BellmanFordSP(G, 0);
      if (spt.hasNegativeCycle())
      {
         double stake = 1000.0;
         for (DirectedEdge e : spt.negativeCycle())
         {
            StdOut.printf("%10.5f %s ", stake, name[e.from()]);
            stake *= Math.exp(-e.weight());
            StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
         }
      }
      else StdOut.println("No arbitrage opportunity");
   }
}
```

This `BellmanFordSP` client finds an arbitrage opportunity in a currency exchange table by constructing a complete-graph representation of the exchange table and then using the Bellman-Ford algorithm to find a negative cycle in the graph.

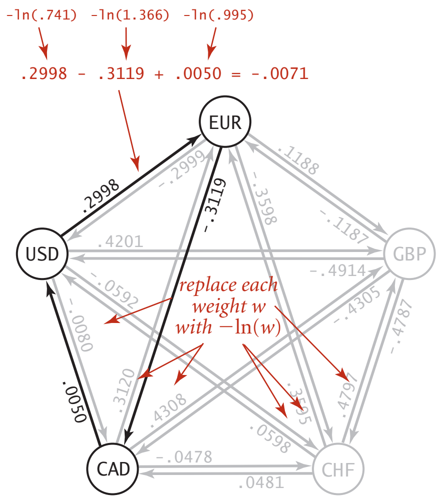
```
% java Arbitrage < rates.txt
1000.00000 USD =  741.00000 EUR
 741.00000 EUR = 1012.20600 CAD
1012.20600 CAD = 1007.14497 USD
```

much, but a currency trader might have 1 million dollars and be able to execute these transactions every minute, which would lead to profits of over \$7,000 per minute, or \$420,000 per hour! This situation is an example of an *arbitrage* opportunity that would allow traders to make unlimited profits were it not for forces outside the model, such as transaction fees or limitations on the size of transactions. Even with these forces, arbitrage is plenty profitable in the real world. What does this problem have to do with shortest paths? The answer to this question is remarkably simple:

Proposition Z. The arbitrage problem is a negative-cycle-detection problem in edge-weighted digraphs.

Proof: Replace each weight by its *logarithm*, negated. With this change, computing path weights by multiplying edge weights in the original problem corresponds to adding them in the transformed problem. Specifically, any product $w_1 w_2 \ldots w_k$ corresponds to a sum $-\ln(w_1) - \ln(w_2) - \ldots - \ln(w_k)$. The transformed edge weights might be negative or positive, a path from v to w gives a way of converting from currency v to currency w, and any negative cycle is an arbitrage opportunity.

In our example, where all transactions are possible, the digraph is a complete graph, so any negative cycle is reachable from any vertex. In general commodity exchanges, some edges may be absent, so the one-argument constructor described in EXERCISE 4.4.43 is needed. No efficient algorithm for finding the *best* arbitrage opportunity (the most negative cycle in a digraph) is known (and the graph does not have to be very big for this computational burden to be overwhelming), but the fastest algorithm to find *any* arbitrage opportunity is crucial—a trader with that algorithm is likely to systematically wipe out numerous opportunities before the second-fastest algorithm finds any.



A negative cycle that represents an arbitrage opportunity

THE TRANSFORMATION IN THE PROOF of PROPOSITION Z is useful even in the absence of arbitrage, because it reduces currency conversion to a shortest-paths problem. Since the logarithm function is monotonic (and we negated the logarithms), the product is maximized precisely when the sum is minimized. The edge weights might be negative or positive, and a shortest path from v to w gives a best way of converting from currency v to currency w.

Perspective The table below summarizes the important characteristics of the shortest-paths algorithms that we have considered in this section. The first reason to choose among the algorithms has to do with basic properties of the digraph at hand. Does it have negative weights? Does it have cycles? Does it have negative cycles? Beyond these basic characteristics, the characteristics of edge-weighted digraphs can vary widely, so choosing among the algorithms requires some experimentation when more than one can apply.

algorithm	restriction	path length compares (order of growth)		extra space	sweet spot
		typical	worst case		
Dijkstra (eager)	positive edge weights	$E \log V$	$E \log V$	V	worst-case guarantee
topological sort	edge-weighted DAGs	$E + V$	$E + V$	V	optimal for acyclic
Bellman-Ford (queue-based)	no negative cycles	$E + V$	VE	V	widely applicable

Performance characteristics of shortest-paths algorithms

Historical notes. Shortest-paths problems have been intensively studied and widely used since the 1950s. The history of Dijkstra's algorithm for computing shortest paths is similar (and related) to the history of Prim's algorithm for computing the MST. The name *Dijkstra's algorithm* is commonly used to refer both to the abstract method of building an SPT by adding vertices in order of their distance from the source and to its implementation as the optimal algorithm for the adjacency-matrix representation, because E. W. Dijkstra presented both in his 1959 paper (and also showed that the same approach could compute the MST). Performance improvements for sparse graphs are dependent on later improvements in priority-queue implementations that are not specific to the shortest-paths problem. Improved performance of Dijkstra's algorithm is one of the most important applications of that technology (for example, with a data structure known as a *Fibonacci heap*, the worst-case bound can be reduced to $E + V \log V$). The Bellman-Ford algorithm has proven to be useful in practice and has found wide application, particularly for general edge-weighted digraphs. While the running time of the Bellman-Ford algorithm is likely to be linear for typical applications, its worst-case running time is VE. The development of a worst-case linear-time shortest-paths algorithm for sparse graphs remains an open problem. The basic Bellman-Ford algorithm