- Dijkstra's algorithm
- implementation
- **negative weights**

Shortest paths application: Currency conversion

Currency conversion. Given currencies and exchange rates, what is best way to convert one ounce of gold to US dollars?

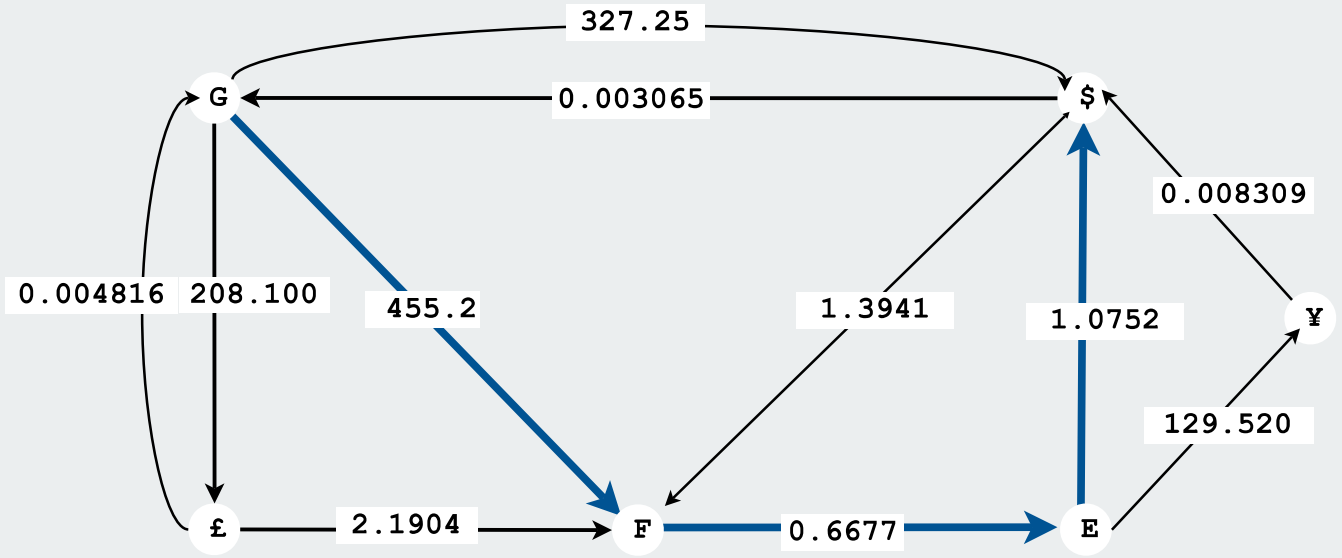
- 1 oz. gold ⇒ $327.25.
- 1 oz. gold ⇒ £208.10 ⇒ ⇒ $327.00. [208.10 × 1.5714]
- 1 oz. gold ⇒ 455.2 Francs ⇒ 304.39 Euros ⇒ $327.28. [455.2 × .6677 × 1.0752]

Currency	£	Euro	¥	Franc	$	Gold
UK Pound	1.0000	0.6853	0.005290	0.4569	0.6368	208.100
Euro	1.4599	1.0000	0.007721	0.6677	0.9303	304.028
Japanese Yen	189.050	129.520	1.0000	85.4694	120.400	39346.7
Swiss Franc	2.1904	1.4978	0.011574	1.0000	1.3941	455.200
US Dollar	1.5714	1.0752	0.008309	0.7182	1.0000	327.250
Gold (oz.)	0.004816	0.003295	0.0000255	0.002201	0.003065	1.0000

Shortest paths application: Currency conversion

Graph formulation.

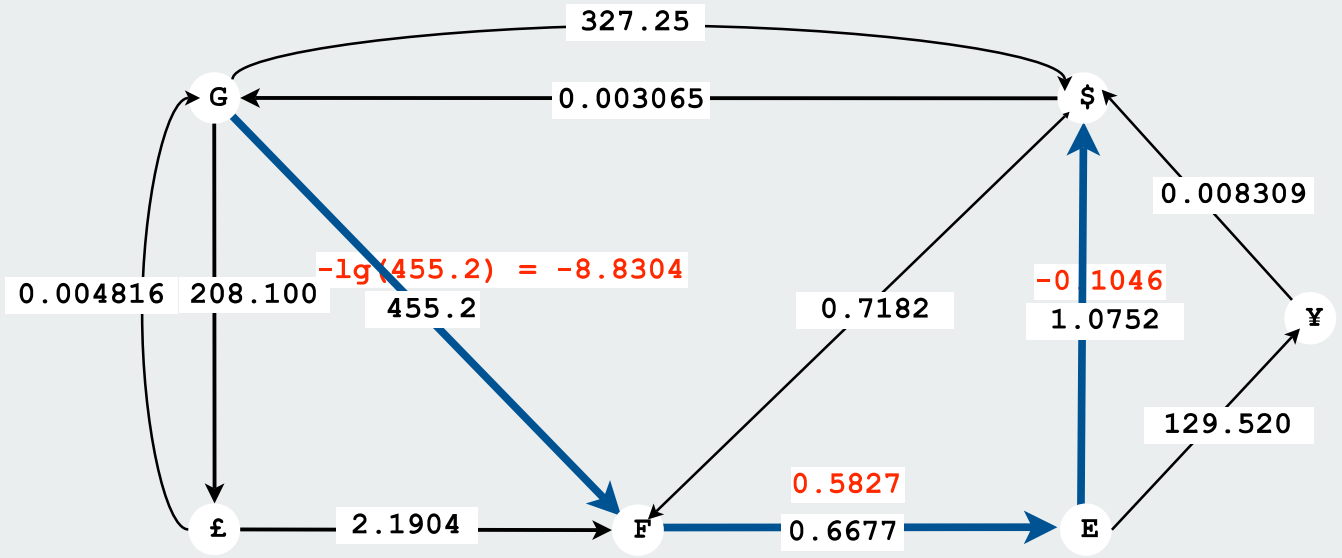
- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find path that maximizes product of weights.



Shortest paths application: Currency conversion

Reduce to shortest path problem by taking logs

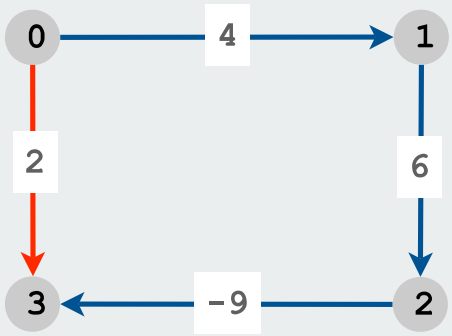
- Let weight(v-w) = – lg (exchange rate from currency v to w)
- multiplication turns to addition
- Shortest path with costs c corresponds to best exchange sequence.



Challenge. Solve shortest path problem with negative weights.

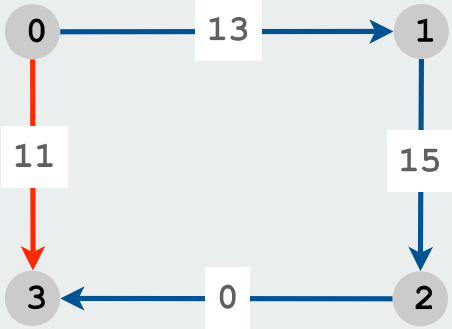
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Adding a constant to every edge weight also doesn't work.

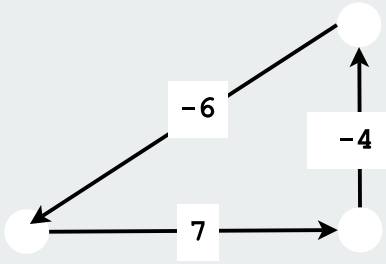


Adding 9 to each edge changes the shortest path because it adds 9 to each segment, wrong thing to do for paths with many segments.

Bad news: need a different algorithm.

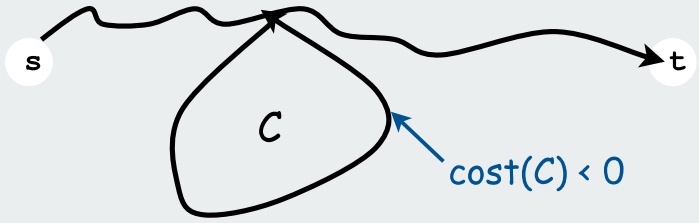
Shortest paths with negative weights: negative cycles

Negative cycle. Directed cycle whose sum of edge weights is negative.



Observations.

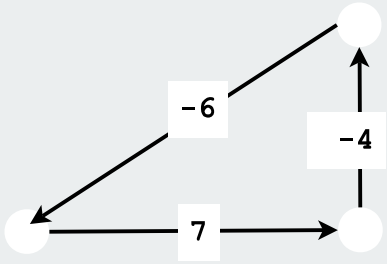
- If negative cycle C on path from `s` to `t`, then shortest path can be made arbitrarily negative by spinning around cycle
- There exists a shortest `s-t` path that is simple.



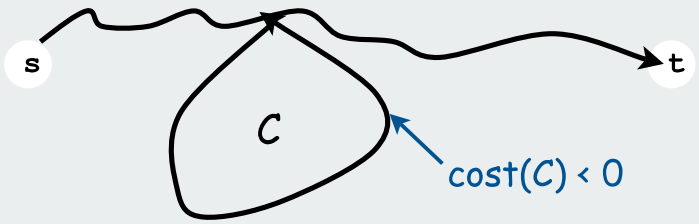
Worse news: need a different problem

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

Good news: Same algorithm solves problem 2 if no negative cycle

Bellman-Ford algorithm

- detects a negative cycle if any exist
- finds shortest simple path if no negative cycle exists

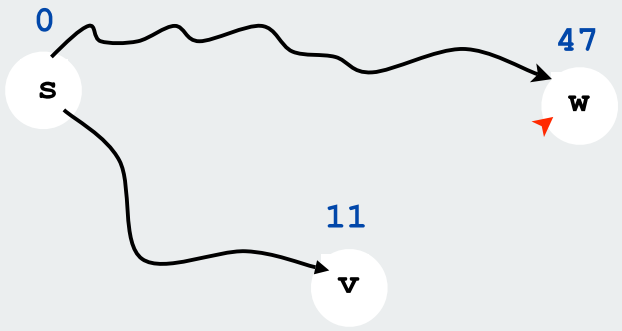
Edge relaxation

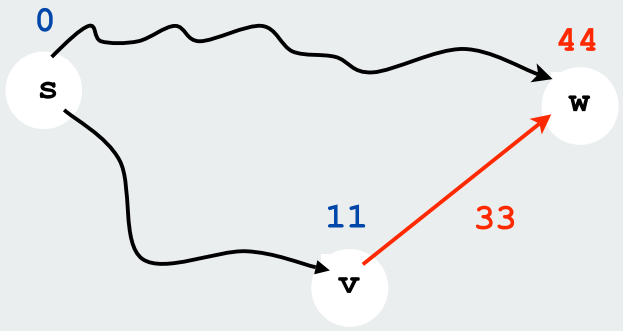
For all `v`, `dist[v]` is the length of some path from `s` to `v`.

Relaxation along edge `e` from `v` to `w`.

- `dist[v]` is length of some path from `s` to `v`
- `dist[w]` is length of some path from `s` to `w`
- if `v-w` gives a shorter path to `w` through `v`, update `dist[w]` and `pred[w]`

```
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight();
    pred[w] = e;
}
```





Relaxation sets `dist[w]` to the length of a shorter path from `s` to `w` (if `v-w` gives one)

Shortest paths with negative weights: dynamic programming algorithm

A simple solution that works!

- Initialize `dist[v] = ∞, dist[s]= 0.`
- Repeat V times: relax each edge `e`.

phase i

```
for (int i = 1; i <= G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (Edge e : G.adj(v))
      {
         int w = e.to();
         if (dist[w] > dist[v] + e.weight())
         {
             dist[w] = dist[v] + e.weight())
             pred[w] = e;
         }
      }
```

relax v-w

Shortest paths with negative weights: dynamic programming algorithm

Running time proportional to E V

Invariant. At end of phase `i`, `dist[v]` ≤ length of any path from `s` to `v` using at most `i` edges.

Theorem. If there are no negative cycles, upon termination `dist[v]` is the length of the shortest path from from `s` to `v`.

and `pred[]` gives the shortest paths

Shortest paths with negative weights: Bellman-Ford-Moore algorithm

Observation. If `dist[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.

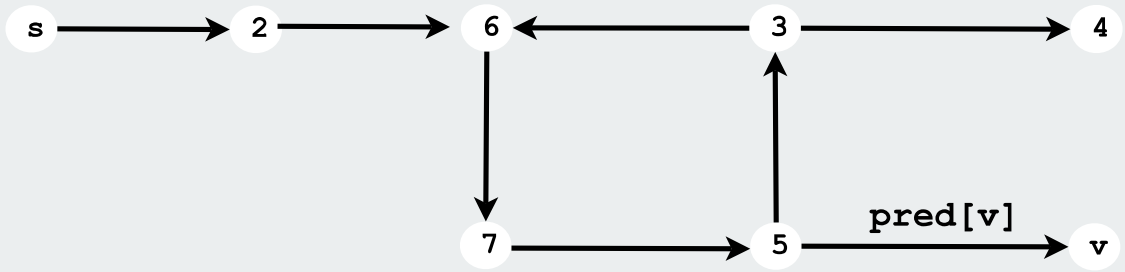
be careful to keep at most one copy of each vertex on queue

Running time.

- still could be proportional to EV in worst case
- much faster than that in practice

Negative cycle detection

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



Finding a negative cycle. If any vertex `v` is updated in phase `v`,
there exists a negative cycle, and we can trace back `pred[v]` to find it.

Negative cycle detection

Goal. Identify a negative cycle (reachable from any vertex).

Solution. Add 0-weight edge from artificial source `s` to each vertex `v`.
Run Bellman-Ford from vertex `s`.