Shortest Paths allowing negative-cost edges

Shortest path problem:
**Given:** a directed graph $G = (V, E)$ with edge weights $c_{vw}$ (possibly negative) and vertices $s, t \in V$.
**Find:** a shortest path in $G$ from $s$ to node $t$.

**Sample Application:** Nodes represent agents in a financial setting and $c_{vw}$ is cost of a transaction in which we buy from agent $v$ and sell immediately to $w$. 
Shortest Paths: Failed Attempts

Why not Dijkstra’s Algorithm? Can fail if negative edge costs.

Dijkstra begins with $S = \{s\}$ and $d(s) = 0$. Next step would add $t$ to $s$ at distance 1, though actual minimum distance from $s$ to $t$ is $-1$.

Adding a constant to every edge cost to make them $\geq 0$? Also fails.

**Problem:** Paths can have different lengths so adding a fixed amount per edge changes relative costs.

Original shortest path is $s$-$v$-$w$-$t$ with cost 3.

After adjustment, shortest path is $s$-$u$-$t$. 
Shortest Paths: Negative Cost Cycles

Negative cost cycle:

Observation: (1) If some path from \( s \) to \( t \) contains a negative cost cycle, there does not exist a shortest \( s-t \) path.

The path can go around the cycle \( W \) more times and get even lower cost, the limit of path costs is \( -\infty \).
Shortest Paths: Negative Cost Cycles

Observation: (1) If some path from \( s \) to \( t \) contains a negative cost cycle, there does not exist a shortest \( s-t \) path.

(2) If the graph \( G \) has no negative cycles then a shortest \( s-t \) path must have at most \( n - 1 \) edges.

If not, there would be a repeated vertex which would create a cycle that could be removed without decreasing the cost.

\[ c(W) \geq 0 \]
Shortest Paths: Dynamic Programming

Defn: $\text{OPT}(i, v) = \text{length of shortest } v \rightarrow t \text{ path } P \text{ using at most } i \text{ edges.}$

Case 1: $P$ uses at most $i - 1$ edges.

• In this case $\text{OPT}(i, v) = \text{OPT}(i - 1, v)$

Case 2: $P$ uses exactly $i$ edges.

• if $(v, w)$ is first edge, then $\text{OPT}$ uses $(v, w)$, and then selects the best $v \rightarrow t$ path using at most $i - 1$ edges

$$\text{OPT}(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min(\text{OPT}(i - 1, v), \min_{(v, w) \in E} c_{vw} + \text{OPT}(i - 1, w)) & \text{otherwise}
\end{cases}$$

By observation: if no negative cost cycles, $\text{OPT}(n - 1, v) = \text{length of shortest } v \rightarrow t \text{ path.}$
Shortest Paths: Implementation

Shortest-Path(G, t) {
    foreach node v ∈ V
        OPT[0, v] ← ∞
        OPT[0, t] ← 0

    for i = 1 to n-1
        foreach node v ∈ V
            OPT[i, v] ← OPT[i-1, v]
        foreach edge (v, w) ∈ E
            OPT[i, v] ← min { OPT[i, v], cvw + OPT[i-1, w] }  
}

To find the shortest paths, maintain a “successor” pointer for each vertex that gives the next vertex on the current shortest path to t.

n − 1 iterations of outer loop
Two inner loops together touch each directed edge once

Total: $O(nm)$ time
$O(n^2)$ space
Shortest Paths: Practical Improvements

Practical improvements:

• Maintain only one array $\text{OPT}[v] = \text{shortest } v-t \text{ path that we have found so far.}$
• No need to check edges of the form $(v, w)$ unless $\text{OPT}[w]$ changed in previous iteration.

**Theorem:** Throughout the algorithm, $\text{OPT}[v]$ is length of some $v-t$ path, and after $i$ rounds of updates, the value $\text{OPT}[v]$ is no larger than the length of shortest $v-t$ path using at most $i$ edges.

Overall impact.

Space: $O(m + n)$.

Running time: Still $O(mn)$ worst case, but substantially faster in practice.
Bellman-Ford: Efficient Implementation

Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        OPT[v] ← ∞
        successor[v] ← φ
    }
    OPT[t] = 0; oldupdated ← {t}
    for i = 1 to n-1 {
        updated ← φ
        foreach node w ∈ V {
            if (w is in oldupdated) {
                foreach node v such that (v, w) ∈ E {
                    if (OPT[v] > c_{vw} + OPT[w]) {
                        OPT[v] ← c_{vw} + OPT[w]
                        successor[v] ← w
                        updated ← updated U {v}
                    }
                }
            }
        }
        if updated = φ, stop.
        else oldupdated ← updated
    }
}
Bellman-Ford
Bellman-Ford
Bellman-Ford
Bellman-Ford
Bellman-Ford
Bellman-Ford
Bellman-Ford
Shortest paths with negative costs on a DAG

Edges only go from lower to higher-numbered vertices

- Update distances in reverse order of topological sort
- Only one pass through vertices required
- $O(n + m)$ time
Distance Vector Protocol
Bellman-Ford Application: Distance Vector Protocol

Application domain: Communication networks
- Node ≈ router
- Edge ≈ direct communication link
- Cost of edge ≈ delay on link.

Edge costs are non-negative, why not use Dijkstra's algorithm?
- Dijkstra’s algorithm requires global information in the network

Advantages of Bellman-Ford approach:
- It only uses only local knowledge of neighboring nodes.
- No need for synchronization: We don't expect routers to run in lockstep. The order in which each `foreach` loop executes is not important. Moreover, the Bellman-Ford algorithm still converges even if updates are asynchronous!
Distance Vector Protocol

**Distance vector protocol:**

- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
- **Algorithm:** each router performs $n$ separate computations, one for each potential destination node.
- “Routing by rumor.”

**Examples:** RIP, Xerox XNS RIP, Novell's IPX RIP, Cisco's IGRP, DEC's DNA Phase IV, AppleTalk’s RTMP.

**Caveat:** Edge costs may *change* during algorithm (or fail completely).

![Diagram](image)
Path Vector Protocols

Link state routing:

- Each router also stores the entire path.
- Based on Dijkstra's algorithm.
- Avoids "counting-to-infinity" problem and related difficulties.
- Requires significantly more storage.

Examples: Border Gateway Protocol (BGP), Open Shortest Path First (OSPF).
Negative Cycles in a Graph
Detecting Negative Cycles

Lemma: If every vertex in $G$ can reach $t$ and $\text{OPT}(n, v) = \text{OPT}(n - 1, v)$ for all $v$, then $G$ has no negative cycles.

Proof: This would be a fixed point of recurrence that computes $\text{OPT}(i, v)$ correctly for every $i$. Vertices on negative cycles that can reach $t$ couldn’t possibly have a fixed point. ■

Lemma: If $\text{OPT}(n, v) < \text{OPT}(n - 1, v)$ for some $v$, then shortest path from $v$ to $t$ with length $\leq n$ contains a cycle $W$. Moreover $W$ has negative cost.

Proof: (By contradiction)

Since $\text{OPT}(n, v) < \text{OPT}(n - 1, v)$, paths $P$ with cost $\text{OPT}(n, v)$ have exactly $n$ edges.

By pigeonhole principle, such a $P$ must contain a directed cycle $W$.

Deleting $W$ yields a $v$-$t$ path with $< n$ edges $\Rightarrow W$ has negative cost.
Detecting Negative Cycles

**Theorem:** Can detect negative cost cycles in $O(mn)$ time.

**Algorithm:** Add new node $t$ and connect all nodes to $t$ with 0-cost edge.

Check if $\text{OPT}(n, v) = \text{OPT}(n - 1, v)$ for all vertices $v$

- if yes, then no negative cycles
- if no, then extract cycle from shortest path from $v$ to $t$
Detecting Negative Cycles: Application

Currency conversion: Given $n$ currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark: High speed trading makes fastest algorithm very valuable!
Detecting Negative Cycles: Summary

Run Bellman-Ford on graph with

• extra node $t$.
• early stopping for up to $n$ iterations (instead of $n - 1$).
• successor variables

Fact: upon termination, successor variables trace a negative cycle if one exists...
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles
Bellman-Ford for Negative Cycles