COMPSCI 311: Introduction to Algorithms
Lecture 16: Dynamic Programming – Shortest Paths
Marius Minea
University of Massachusetts Amherst
slides credit: Dan Sheldon
25 March 2019

Currency Trading
Given: directed graph with exchange rate $r_e$ on edge $e$
Find best exchange rate $s \rightarrow t$,
i.e., path $P$ with maximum product $\prod_{e \in P} r_e$ over edges
Assumption (no arbitrage): no cycles $C$ with $\prod_{e \in C} r_e > 1$.

Compute optimal path cost, but
- product, not sum
- maximum, not minimum

From Rates to Costs
- From product to sum: take logarithm!
  - logarithm of product is sum of logs
- Maximize $x$ means minimize $-x$
- Let $c_e = -\log r_e$ be the cost of edge $e$
  - Highest rate path is now minimum cost path

Reduce to Shortest Paths
- Define $\text{cost}(P)$ to be the negative log of its exchange rate.
  Then the highest rate path is now the lowest cost path.
- But $\text{cost}(P)$ is also the sum of its edge costs:
  $\text{cost}(P) = -\log \prod_{e \in P} r_e$
  $= \sum_{e \in P} (-\log r_e)$
  $= \sum_{e \in P} c_e$
  - Equivalent problem: find the $s \rightarrow t$ path of minimum cost

Currency Trading with Shortest Paths
- Negative edge weights!
  - Edge costs are now $c_e = -\log r_e$
- Problem: given a graph with possibly negative edge weights, find shortest $s \rightarrow t$ path
- Assumption: no cycle $C$ with $\sum_{e \in C} c_e < 0$. Why?

Dijkstra’s Algorithm: Negative Edge Behavior
What is the shortest path value the algorithm finds for $d(s, v)$?
Bellman-Ford Algorithm: Setup

Consider shortest paths from any node to a given target node \( t \) (single-destination shortest paths).

Like single-source, but destination more relevant e.g., in routing

- Dijkstra’s algorithm started with closest neighbor path must be edge, can’t get shorter
- Not true for negative costs: can keep decreasing
- Need different order: increasing edge count to target \( t \)

**Fact.** If no negative cycles, shortest path has at most \( n - 1 \) edges. Why?
Path with \( \geq n \) edges has \( \geq n + 1 \) nodes: would repeat some node, thus have a cycle. Can “cut out” nonnegative cycle for shorter path.

Towards a Recurrence

For shortest paths from any \( v \) to a fixed \( t \), we’d like to compute \( \text{OPT}(i + 1, v) \) from \( \text{OPT}(i, v) \), by incrementing the edge count \( i \).

If we find a better \( v \rightarrow t \) path starting with edge \( (v, w) \), we’ll update
\[ \text{OPT}(i + 1, v) = c_{v,w} + \text{OPT}(i, w) \]

Should \( \text{OPT}(i, v) \) mean the optimal cost from \( v \) to \( t \):

- on a path with exactly \( i \) edges?
- on a path with at most \( i \) edges?

In the end, want at most \( n - 1 \) edges (may be any number)

Bellman-Ford Recurrence

- Let \( \text{OPT}(i, v) \) be cost of shortest \( v \rightarrow t \) path with at most \( i \) edges.
- **Base case:** \( \text{OPT}(0, t) = 0 \), \( \text{OPT}(0, s) = \infty \) for \( s \neq t \)
- **Recurrence:** let \( P \) be the optimal \( v \rightarrow t \) path using at most \( i + 1 \) edges.
  - if \( P \) uses at most \( i \) edges, then \( \text{OPT}(i + 1, v) = \text{OPT}(i, v) \).
  - else \( P = v \rightarrow w \rightarrow t \) where \( w \rightarrow t \) path uses at most \( i \) edges.
    \[ \text{OPT}(i + 1, v) = c_{v,w} + \text{OPT}(i, w) \]

\[ \text{OPT}(i, v) = \min \left\{ \text{OPT}(i - 1, v), \min_{(v,w) \in E} \left\{ c_{v,w} + \text{OPT}(i - 1, w) \right\} \right\} \]

Bellman-Ford Algorithm

\[ \text{OPT}(i, v) = \min \left\{ \text{OPT}(i - 1, v), \min_{(v,w) \in E} \left\{ c_{v,w} + \text{OPT}(i - 1, w) \right\} \right\} \]

**Shortest-Path(\( G, t \))**

- \( n \) = number of nodes in \( G \)
- create array \( M \) of size \( n \times n \) (iterations \( \times \) nodes)
- \( M[0, 0] = 0 \) and \( M[0, v] = \infty \) for all \( v \neq t \)

for \( i = 1 \) to \( n - 1 \) do

for all nodes \( v \neq t \) do

for all \( (v,w) \in E \) do

if \( M[i, v] > c[v,w] + M[i - 1, w] \) then

\[ M[i, v] = c[v,w] + M[i - 1, w] \]

Running time? \( O(n(n + m)) \). If graph connected, \( O(mn) \).
Clicker Question 3

Suppose $M[i, v] = M[i - 1, v]$ for all $v$. Then

A. There are no negative edge costs in the graph.

B. There is a negative cycle in the graph.

C. All $v \rightarrow t$ paths have at most $i$ edges.

D. We can terminate the algorithm after the $i$th iteration, because no future values will change.

Improvements

- Reduce memory $O(n^2) \rightarrow O(n)$
  
  Only need path lengths for $i - 1$ and $i$ (vector, not matrix) can actually just update a distance vector $d[i]$ in-place

- Keep track of path: $\text{succ}[v] = \text{next node on path to } t$
  
  Initially, $\text{succ}[v] = \text{null}$ for all $v \neq t$
  
  When updating, $M[i, v] = c[v, w] + M[i - 1, w]$, set $\text{succ}[v] = w$

- Try updates only when needed

  Update means path of length $i$, thus $w$ was updated in step $i - 1$.
  
  Keep track of nodes $w$ updated at each step
  
  Next step, only try to update their predecessors

Analysis

- Does following $\text{succ}[v]$ links get us path of length $d[v]$?
  
  No, might be shorter, if $d[v]$ updated one step later

- Does following successor links always lead to target $t$?
  
  Yes, if and only if there is no negative-length cycle

- How to detect negative-length cycles?
  
  Run algorithm for one extra step!

Detecting Negative-Weight Cycles

If no negative-weight cycles, shortest path has $\leq n - 1$ edges.

If some $d[v]$ decreases in $n^{th}$ iteration $\Rightarrow$ negative-weight cycle!

But this is only over paths to a fixed target node $t$.

How to cover the entire graph? And find the actual cycle?

Add dummy sink node with zero-cost edges from all nodes.

Use this as target (all nodes are predecessors and will be covered).

Still $O(n)$ space, $O(mn)$ time.
Finding Negative-Weight Cycles Early

Do we need to wait for the $n$th iteration?

If no cycles, $\text{succ}[]$ pointers form a tree leading to root $t$.
Suppose we update $\text{succ}[v] = w$. Two ways to check for new cycle:

- Follow pointers from $w$, looking for $v$. Bad, could be $O(n)$.
- Store tree rooted at $v$ (list of all nodes $x$ with $\text{succ}[x] = v$).
  Recursively check whether $w$ is in tree of $v$.

Insight: Check takes time proportional to work already done
(setting up the $\text{succ}[]$ pointers).

Careful: claim credit for work done only once (or constant times).
⇒ while checking $w$, remove all nodes from tree of $v$.
Since they have paths to $v$ and $d[v]$ updated, they’ll be added again.

Shortest-path complexity preserved: $O(n)$ space, $O(mn)$ time.
Negative-weight cycle $c$ found after $\text{length}(c)$ iterations.