COMPSCI 311: Introduction to Algorithms

Lecture 16: Dynamic Programming – Shortest Paths

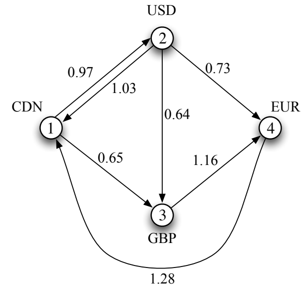
Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

25 March 2019

Currency Trading



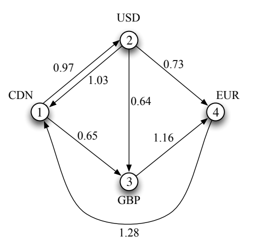
- **Given**: directed graph with exchange rate r_e on edge e
- **Find** best exchange rate $s \to t$, i.e., path P with maximum product $\prod_{e \in P} r_e$ over edges
- **Assumption** (no arbitrage): no cycles C with $\prod_{e \in C} r_e > 1$.

Compute optimal path cost, but

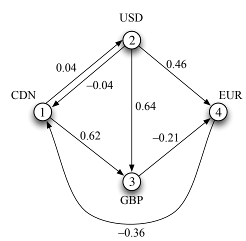
- product, not sum
- maximum, not minimum

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



Rates



Costs

- Highest *rate* path is now minimum *cost* path

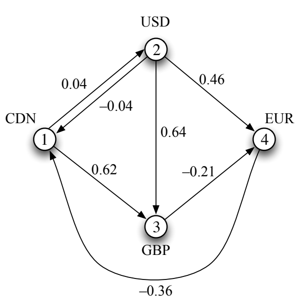
Reduce to Shortest Paths

- 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 cost(P) is also the sum of its edge costs:

$$\begin{aligned}\text{cost}(P) &= -\log \prod_{e \in P} r_e \\ &= \sum_{e \in P} (-\log r_e) \\ &= \sum_{e \in P} c_e\end{aligned}$$

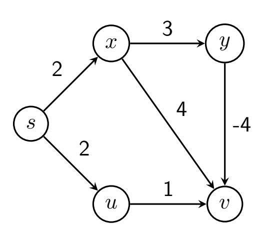
- Equivalent problem: find the $s \to t$ path of minimum cost

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 $s \to t$ 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 $d(s, v)$?

Clicker Question 1

When run on a graph with negative edges, Dijkstra's algorithm:

A. Does not give the right value if shortest path has negative edge.

B. May give the right value even if shortest path has a negative edge.

C. Does not give the right value if the target node is first reached through a positive edge.

D. Gives the right value if the target node is first reached through a negative edge.

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.

Clicker Question 2

In a directed graph with $n+2$ nodes, the maximum number of acyclic paths from a node s to a node $t \neq s$ is:

A. $\leq 2^n$

B. $\leq (n-1)!$

C. $\leq n!$

D. can be $> n!$

Towards a Recurrence

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

If we find a better $v \rightsquigarrow t$ path starting with edge (v, w), we'll update

$\mathrm{OPT}(i+1, v) = c_{v,w} + \mathrm{OPT}(i, w)$

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

$$\mathrm{OPT}(i, v) = \min\left\{\mathrm{OPT}(i-1, v),\quad \min_{(v,w)\in E}\{c_{v,w} + \mathrm{OPT}(i-1, w)\}\right\}$$

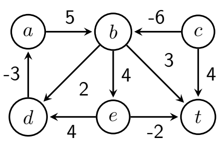
Bellman-Ford Algorithm

$$\mathrm{OPT}(i, v) = \min\left\{\mathrm{OPT}(i-1, v),\quad \min_{(v,w)\in E}\{c_{v,w} + \mathrm{OPT}(i-1, w)\}\right\}$$

Shortest-Path(G, t)
  n = number of nodes in G
  create array M of size $n \times n$ (iterations $\times$ nodes)
  set $M[0, t] = 0$ and $M[0, v] = \infty$ for all $v \neq t$
  for $i = 1$ to $n-1$ **do** ▷ $n-1$ times
    for all nodes $v \neq t$ **do** ▷ $n-1$ times
      $M[i, v] = M[i-1, v]$ ▷ less than i edges
      for all $(v, w) \in E$ **do**
        if $M[i, v] > c[v, w] + M[i-1, w]$ **then** ▷ m times
          $M[i, v] = c[v, w] + M[i-1, w]$

Running time? $O(n(n+m))$. If graph connected, $O(mn)$.

Example



	a	b	c	d	e	t
0:	∞	∞	∞	∞	∞	0
1:	∞	3	4	∞	-2	0
2:	8	2	-3	∞	-2	0
3:	7	2	-4	5	-2	0
4:	7	2	-4	4	-2	0
5:	7	2	-4	4	-2	0

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 $v \rightsquigarrow t$ paths have at most i edges.

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

Improvements

- Reduce memory $O(n^2) \to O(n)$

Only need path lengths for $i-1$ and i (vector, not matrix)
can actually just update a distance vector $d[]$ in-place

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

- 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

Bellman-Ford-Moore: Efficient Implementation

Shortest-Path(G, t)
  set $d[t] = 0$ and $d[v] = \infty$ for all $v \neq t$
  set $succ[v] = null$ for all v
  for $i = 1$ to $n-1$ **do**
    for all nodes $w \neq t$ **do**
      if w updated in previous pass **then**
        for all $(v, w) \in E$ **do**
          if $d[v] > c[v, w] + d[w]$ **then**
            $d[v] = c[v, w] + d[w]$
            $succ[v] = w$

Analysis

- Does following $succ[v]$ 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

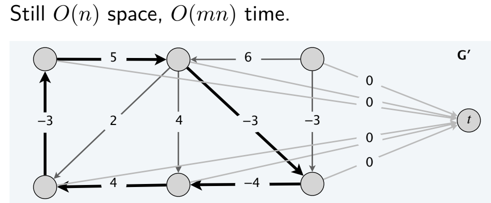
If no negative-weight cycles, shortest path has $\leq n-1$ edges.

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, $succ[]$ pointers form a tree leading to root t.
Suppose we update $succ[v] = w$. 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 $succ[x] = v$).
  Recursively check whether w is in tree of v.

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

Careful: claim credit for work done only **once** (or constant times).
$\Rightarrow$ 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 *length(c)* iterations.