Week 10.1, Monday, Oct 21

Homework 5 Due October 26 @ 11:59PM on Gradescope
6.8 Shortest Paths (with Negative Weights)
Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$. 
For which of the following directed graphs will Dijkstra find the shortest path from s to t?

A

B

C

D

E: None of the above
For which of the following directed graphs will Dijkstra find the shortest path from s to t?

A

\[ \begin{array}{c}
\text{s} \\
\downarrow 1 \\
\text{t} \\
\downarrow -6 \\
\text{u} \\
\downarrow 2 \\
\text{v} \\
\end{array} \]

B

\[ \begin{array}{c}
\text{s} \\
\downarrow 1 \\
\text{t} \\
\downarrow -6 \\
\text{u} \\
\downarrow 2 \\
\text{v} \\
\end{array} \]

C

\[ \begin{array}{c}
\text{s} \\
\downarrow 3 \\
\text{t} \\
\downarrow -6 \\
\text{u} \\
\downarrow 4 \\
\text{v} \\
\end{array} \]

D

\[ \begin{array}{c}
\text{s} \\
\downarrow 3 \\
\text{t} \\
\downarrow -6 \\
\text{u} \\
\downarrow 10 \\
\text{v} \\
\end{array} \]

E: None of the above
Shortest Paths: Dynamic Programming

**Def.** $OPT(i, v) = \text{length of shortest } v-t \text{ path } P \text{ using at most } i \text{ edges.}$

- Case 1: $P$ uses at most $i-1$ edges.
  - $OPT(i, v) = OPT(i-1, v)$

- Case 2: $P$ uses exactly $i$ edges.
  - if $(v, w)$ is first edge, then $OPT$ uses $(v, w)$, and then selects best $w-t$ path using at most $i-1$ edges

**Remark.** By previous observation, if no negative cycles, then $OPT(n-1, v) = \text{length of shortest } v-t \text{ path.}$

**Fact:** If there is a negative cycle then $OPT(n,v) < OPT(n-1,v)$ for some node $v$
Shortest Paths: Implementation

\[
\text{Shortest-Path}(G, t) \{
\text{foreach node } v \in V \\
M[0, v] \leftarrow \infty \\
M[0, t] \leftarrow 0 \\
\text{for } i = 1 \text{ to } n-1 \\
\text{foreach node } v \in V \\
M[i, v] \leftarrow M[i-1, v] \\
\text{foreach edge } (v, w) \in E \\
M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \}
\}
\]

Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space.

Finding the shortest paths. Maintain a "successor" for each table entry i.e. if $(v,w)$ is the first edge on the shortest $i$-edge path $P$ from $v$ to $t$ then $\text{Successor}[v] = w$
Bellman-Ford: Efficient Implementation

Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← φ
    }

    M[t] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
            if (M[w] has been updated in previous iteration) {
                foreach node v such that (v, w) ∈ E {
                    if (M[v] > M[w] + c_{vw}) {
                        M[v] ← M[w] + c_{vw}
                        successor[v] ← w
                    }
                }
            }
        }

        If no M[w] value changed in iteration i, stop.
    }
}
Practical improvements.

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

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

Overall impact.

- Memory: $O(n)$ additional memory beyond $O(m+n)$ for input (adjacency list).
- Running time: $O(mn)$ worst case, but substantially faster in practice.

Detect Negative Cycle.

- Run Outer Loop for $n$ iterations (for $i = 1$ to $n-1 \ n$)
- If any $M[v]$ changes in iteration $n \Rightarrow$ negative cycle
All Pairs Shortest Path

**Input:**
Graph $G=(V,E)$, directed and weighted, with weights $w(e)$

**Output:**
Shortest path matrix $D$, where $d(u,v)$ represents the cost of the shortest paths from $u$ to $v$
- The vertices on a shortest path are typically generated on a need basis

One solution: solve $n$ single-source problems
- No negative weights: $O(nm + n^2 \log n)$ time using Dijkstra
How do we get started on a dynamic programming formulation?

- We compute $n^2$ entries of matrix $D$
- We do not know how many edges the shortest path from $u$ to $v$ contains
- We do not know in what order vertices are visited
- The principle of optimality holds for subpaths in a shortest path

How do we build up solutions in a systematic way?
A First DP Solution

Input is adjacency matrix $A$; no negative cycles

\[ d(i,j)^r = \text{cost of shortest path from } i \text{ to } j \]

using at most $r$ edges

We know

- $d(i,i)^0 = 0$ and $d(i,j)^0 = \infty$ for $i \neq j$
- determine $d(i,j)^r$ from earlier computed values
- $d(i,j)^{n-1}$ represent the shortest paths
\[ d(i, j)^r = \min_{k: (k, j) \in E} \{ d(i, k)^{r-1} + w(k, j) \} \]

O\((n^2m)\) time to fill in DP table
Floyd-Warshall algorithm

Define $c_{ij}^{(k)} = \text{weight of a shortest path from } i \text{ to } j$
with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.

Thus, $\delta(i, j) = c_{ij}^{(n)}$. 
Floyd-Warshall all-pair shortest paths

Input is an adjacency matrix $A$

$c(i,j)^k = \text{cost of the shortest path from } i \text{ to } j \text{ with intermediate vertices belonging to set } \{1, 2, 3, \ldots, k\}$

- $c(i,j)^0 = A(i,j)$
- $c(i,j)^n$ is the final answer

Recursive formulation:

$c(i,j)^k = \min \{ c(i,j)^{k-1}, c(i,k)^{k-1} + c(k,j)^{k-1} \}$

$O(n^3)$ time algorithm
Floyd-Warshall recurrence

\[ c_{ij}^{(k)} = \min_k \{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \} \]

Case 1: path \( P \) only uses intermediate vertices from \( \{1, \ldots, k-1\} \)

Case 2: path \( P \) includes node \( k \)

\( P := \) shortest path from \( i \) to \( j \) such that intermediate vertices are in set \( \{1, 2, \ldots, k\} \)
Pseudocode for Floyd-Warshall

for $k \leftarrow 1$ to $n$
  do for $i \leftarrow 1$ to $n$
    do for $j \leftarrow 1$ to $n$
      do if $c_{ij} > c_{ik} + c_{kj}$
        then $c_{ij} \leftarrow c_{ik} + c_{kj}$

• Can drop the superscripts (extra relaxations can’t hurt)
• $\Theta(n^3)$ time.
• Simple to code.