Notes, 7(d)

ECE 606

Floyd-Warshall

A final example we discuss in Lecture 7 on dynamic programming is an ingenious algorithm for the all-source shortest distances/paths problem.

Previously, we considered single-source shortest distances/paths only. That is, part of our input is a source-vertex s, and we compute shortest distances and paths to every other vertex from s only. What if we seek shortest distances and paths between every pair of vertices?

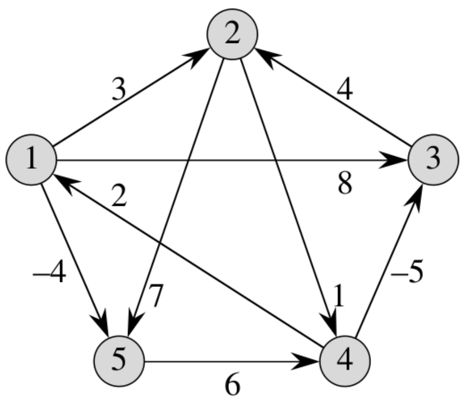
That is, our input is a graph undirected or directed. Suppose the input graph has n vertices. Assume the vertices are named $\{1, 2, \ldots, n\}$. Then, our output is two $n \times n$ matrices, D and Π. $D[i, j]$ is the shortest-distance from vertex i to j; it is the special symbol ∞ if no path $i \rightsquigarrow j$ exists and 0 if $i = j$. $\Pi[i, j]$ is a predecessor of $\cancel{i}$ j in a shortest-path $i \rightsquigarrow j$, i.e., of weight $D[i, j]$; it is the special mnemonic NIL if no path $i \rightsquigarrow j$ exists, or $i = j$.

We could simply run Bellman-Ford with every vertex as the source-vertex. As each such run has time-efficiency $\Theta(VE)$, we would have a time-efficiency of $\Theta(V^2E)$, which is $\Theta(V^4)$ in the worst-case.

If we know that we have non-negative edges only, then we could run Dijkstra repeatedly with each vertex as the source-vertex with, for example, the priority queue implemented using a binary heap. Then, we would have a worst-case time-efficiency of $\Theta(VE \lg V) = \Theta(V^3 \lg V)$.

Floyd-Warshall exploits the optimal substructure of shortest paths the following way. Suppose our vertices are named $\{1, 2, \ldots, n\}$, i.e., $|V| = n$. Define $d_{u,v}^{(k)}$ as the shortest distance from u to v such that paths are allowed to traverse the vertices $\{1, 2, \ldots, k\}$ only. By "traverse," we mean, use as intermediate vertices.

For example, for every $u, v \in V, u \neq v$, $d_{u,v}^{(0)} = w(u, v)$ if $\langle u, v \rangle \in E$ and ∞ otherwise. For more examples, consider the following graph.



$d_{1,4}^{(0)} = d_{1,4}^{(1)} = \infty$. But $d_{1,4}^{(2)} = 4$ on account of the path $1 \to 2 \to 4$. But that is not a shortest-path in the graph from 1 to 4. It turns out $d_{1,4}^{(5)} = 2$ on account of $1 \to 5 \to 4$.

We can write a recurrence for $d_{i,j}^{(k)}$, which then immediately suggests a bottom-up algorithm based on dynamic programming.

First, we specify $w_{i,j}$ for every $i, j \in V$, as a generalization of $w\colon E \to \mathbb{R}$. It is a function, $w_{i,j}\colon V \times V \to \mathbb{R} \cup \{\infty\}$, where:

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } \langle i, j \rangle \in E \\ \infty & \text{otherwise} \end{cases}$$

Now, our recurrence for $d_{i,j}^{(k)}$ is as follows:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0 \\ \min\Big\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \Big\} & \text{otherwise} \end{cases}$$

We can think of the above recurrence as exploiting the fact that $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$ whenever $i = k$ or $j = k$. Because, if we have no negative edge-weight cycles, then every shortest path is simple.

Our corresponding recurrence for $\pi_{i,j}^{(k)}$, a predecessor vertex of i on a path that has weight $d_{i,j}^{(k)}$, is as follows:

$$\pi_{i,j}^{(k)} = \begin{cases} \textsc{Nil} & \text{if } k = 0 \text{ and } w_{i,j} \in \{0, \infty\} \\ i & \text{if } k = 0 \text{ and } w_{i,j} \notin \{0, \infty\} \\ \pi_{i,j}^{(k-1)} & \text{if } k > 0 \text{ and } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)} & \text{otherwise} \end{cases}$$

It should now be easy to write down a bottom-up algorithm that uses dynamic programming. We would initialize $D^{(0)} = [w_{i,j}]$, and then iterate for all k from 1 to n, for all i from 1 to n, and for all j from 1 to n to progressively construct each $D^{(1)}, D^{(2)}, \ldots, D^{(n)}$, and simultaneously, $\Pi^{(k)}$.

At the moment we compute $D^{(k)}$, we need the immediately prior $D^{(k-1)}$ only, and not, for example, $D^{(k-2)}, \ldots, D^{(0)}$. So at any moment, we need to maintain two $n \times n$ matrices only, that correspond to D.

This algorithm's time-efficiency is $\Theta(V^3)$, which is better than both Dijkstra and Bellman-Ford, in the worst-case. Also, it can handle negative edge weights.