CS124	Lecture 5	Spring 2015

1 Bellman-Ford

Let $G = (V, E)$ be a graph with arbitrary edge weights (positive or negative) on n vertices and m edges. We use $w(v, u)$ to denote the edge weight from v to u (by convention we say it is ∞ if the edge does not exist). Lecture notes 4 provides an iterative implementation of the Bellman-Ford algorithm which solves the single-source shortest paths problem on such a graph. One might ask: how could we come up with this algorithm if we were to design it ourselves? The following is the way I personally think about Bellman ford: recursion.

Suppose s is our source vertex that we are finding shortest paths from. Define a function $f(u, k)$ which is the length of the shortest path from s to u using at most k edges. First, let us suppose that we were promised that G has no negative weight cycles. In this case, the length of the shortest path from s to any u would be $f(u, n-1)$, since taking $k > n-1$ would imply a cycle, which cannot provide any benefit if there are no negative weight cycles. So, how can we calculate $f(u, n-1)$? Recursively!

$$f(u,k) = \begin{cases} 0, & \text{if } u = s, k = 0 \\ \infty, & \text{if } u \neq s, k = 0 \\ \min\left\{ f(u,k-1), \min_{v \in V:(v,u) \in E} f(v,k-1) + w(v,u) \right\}, & \text{otherwise} \end{cases}$$

In words, the base case is $k = 0$ (a path with zero edges), in which the length of the shortest path is either 0 or ∞ depending on whether $u = s$ (we also use ∞ as the shortest path distance to represent the lack of a path). If $k > 0$, then the shortest path from s to u of length at most k is, by the law of the excluded middle, either of length less than k or exactly k. Thus we simply take the minimum of the two possibilities. The $f(u, k-1)$ term represents the first possibility. The other term in the minimum, which takes a min over $v \in V$, represents the other possibility: a path of length k is simply a path of length $k-1$, ending at some node v, followed by the edge v, u. Shortest paths have the property that all subpaths are themselves shortest paths (justify this to yourself as an exercise!), and thus the path of length $k-1$ to v should itself be a shortest path. Note that above we actually take $f(v, k-1) + w(v, u)$, and $f(v, k-1)$ isn't actually a path of length exactly $k-1$, but rather simply *at most* $k-1$. This is OK though, since this only makes the minimum smaller and $f(v, k-1) + w(v, u)$ is still a valid upper bound on $f(u, k)$.

As an exercise, you may want to show that the straightforward recursive implementation of computation of $f(u, n-1)$ would take exponential time. This is where a technique called *memoization* comes in (we'll see more of this later when we start doing dynamic programming). The idea behind memoization is to store already-computed answers in a lookup table, so that if later recursive calls want to recompute that answer we can return it from the table immediately. Now note for our problem: there are at most n^2 possibilities for the arguments (u,k) fed into f when doing recursive calls from $f(u,n-1)$. This is because there are n possibilities for each of u and k. So, let us simply store a lookup table (a 2-dimensional array) $\texttt{ans}[][]$ which is n by n, and a $\texttt{seen}[][]$ array of the same size. We initialize the `seen` array to all false. Then when we enter a call to $f(u,k)$, we first check if $\texttt{seen}[u][k]$ is true. If so, it means we've already computed f on this input and stored the answer in the lookup table, so we simply return $\texttt{ans}[u][k]$. Otherwise, we set $\texttt{seen}[u][k]$ to true, compute $f(u,k)$ recursively, then store the answer in $\texttt{ans}[u][k]$. The running time of this memoized version is $O((m+n)n)$. This is because for any fixed u,k for which we have not already precomputed the answer, we spend $indegree(u)$ time looping over $v \in V$ such that $(v,u) \in E$. Thus the total time spent for fixed k (for which there are n possibilities) over all choices of u is $\sum_{u \in U} indegree(u) = m$. We also spend an additional constant time for each u for simple operations like taking minimums of two numbers. Thus we spend $O(n+m)$ time for each k, and thus $O((n+m)n)$ time total over all k's. This is the same time complexity as the iterative Bellman-ford algorithm, although the recursive implementation uses more space: $O(n^2)$ (to store the lookup table) instead of $O(n+m)$ for the iterative implementation. The main reason we save in the iterative implementation is because $\texttt{ans}[u][k]$ only depends on ans$[][k-1]$ for every u,k, so we only need to remember answers for the last value of k in our loop in an iterative implementation.

To check whether there are negative weight cycles, we check to make sure that for all $u \in V$, $f(u,n) \geq f(u,n-1)$. If there are no negative weight cycles this test will clearly hold, since a length n path must contain a cycle, and having no negative weight cycles means that this cannot be advantageous. For the other direction, suppose $f(u,n) \geq f(u,n-1)$ for all u. This is the same as saying $f(u,n-1) \leq f(v,n-1) + w(v,u)$ for all $u,v \in V$. Consider a cycle $v_1, v_2, \ldots, v_\ell$ in the graph. We just need to show this cycle has nonnegative total weight. For each i we have

$$f(v_{i+1}, n-1) \leq f(v_i, n-1) + w(v_i, v_{i+1})$$

where we treat "$\ell + 1$" as 1 (wrapping around the cycle). Summing over all i from 1 to ℓ, we obtain the inequality $\sum_{i=1}^{\ell} w(v_i, v_{i+1}) \geq 0$, as desired.

2 All pairs shortest paths

Let G be a graph with arbitrary edge weights (positive or negative). We want to calculate the shortest paths between *every* pair of nodes. One way to do this is to run Dijkstra's algorithm several times, once for each node. Here we develop a different solution, called the Floyd-Warshall algorithm.

Let us first assume that there are no negative weight cycles. Like with Bellman-Ford above, we can come up with a recursive solution. Let $f(u,v,k)$ be the length of the shortest path from u to v using *only* nodes $1\ldots k$ as intermediate nodes. Of course when k equals the number of nodes in the graph, n, we will have solved the original problem. Again we can compute f recursively:

$$f(u,v,k) = \begin{cases} w(u,v), & \text{if } k=0 \\ \min\{f(u,v,k-1), f(u,k,k-1)+f(k,v,k-1)\}, & \text{otherwise} \end{cases}$$

Again outside the base case we use the law of the excluded middle: either the shortest path when allowed to use intermediate nodes $1,\ldots,k$ uses node k or it doesn't. We simply take the minimum of the two possibilities. There are n^3 states and we do $O(1)$ work for each one (a minimum of two numbers) ignoring recursive calls. Thus with memoization the total running time would be $O(n^3)$. The space naively would unfortunately also be $\Theta(n^3)$, but an iterative implemtntation, which we now describe, does better.

We let the matrix $D_k[i.j]$ represent the length of the shortest path between i and j using intermediate nodes $1\ldots k$. Initially, we set a matrix D_0 with the direct distances between nodes, given by d_{ij}. Then D_k is easily computed from the subproblems D_{k-1} as follows:

$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]).$$

It might seem that we need at least two matrices to code this, but in fact it can all be done in one loop. (**Exercise:** think about it!)

$D = (d_{ij})$, distance array, with weights from all i to all j
for $k = 1$ to n do
    for $i = 1$ to n do
        for $j = 1$ to n do
            $D[i,j] = \min(D[i,j], D[i.k] + D[k,j])$

Note that again we can keep an auxiliary array to recall the actual paths. We simply keep track of the last intermediate node found on the path from i to j. We reconstruct the path by successively reconstructing intermediate nodes, until we reach the ends.

What about if there are negative weight cycles?

Exercise. Show that there is a negative weight cycle in G if and only if for some $1 \leq i \leq n$, $D[i,i] < 0$ at the end of running Floyd Warshall.