## CS 3510 - Design and Analysis of Algorithms

Maria-Florina Balcan Lectures 27 and 28: March 18th and 28th, 2011

---

# Dynamic-Programming algorithms for shortest path problems[1]

We are now going to look to a basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge $(i, j)$ in our graph, let's use $len(i, j)$ to denote its length. The basic shortest-path problem is as follows:

**Definition 1** *Given a weighted, directed graph $G$, a start node $s$ and a destination node $t$, the* **s-t shortest path** *problem is to output the shortest path from $s$ to $t$. The* **single-source** *shortest path problem is to find shortest paths from $s$ to every node in $G$. The (algorithmically equivalent)* **single-sink** *shortest path problem is to find shortest paths from every node in $G$ to $t$.*

We will allow for negative-weight edges (we'll later see some problems where this comes up when using shortest-path algorithms as a subroutine) but will assume no negative-weight cycles (else the shortest path can wrap around such a cycle infinitely often and has length negative infinity). As a shorthand, if there is an edge of length $\ell$ from $i$ to $j$ and also an edge of length $\ell$ from $j$ to $i$, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

## 0.1 The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-sink (or single-source) shortest path problem.[2] Let us develop the algorithm using the following example:

[1] These lecture notes are due to Avrim Blum.

[2] Bellman is credited for inventing Dynamic Programming, and even if the technique can be said to exist inside some algorithms before him, he was the first to distill it as an important technique.

How can we use Dyanamic Programming to find the shortest path from all nodes to $t$? First of all, as usual for Dynamic Programming, let's just compute the *lengths* of the shortest paths first, and afterwards we can easily reconstruct the paths themselves. The idea for the algorithm is as follows:

1. For each node $v$, find the length of the shortest path to $t$ that uses at most 1 edge, or write down $\infty$ if there is no such path.

   This is easy: if $v = t$ we get 0; if $(v, t) \in E$ then we get $len(v, t)$; else just put down $\infty$.

2. Now, suppose for all $v$ we have solved for length of the shortest path to $t$ that uses $i - 1$ or fewer edges. How can we use this to solve for the shortest path that uses $i$ or fewer edges?

   Answer: the shortest path from $v$ to $t$ that uses $i$ or fewer edges will first go to some neighbor $x$ of $v$, and then take the shortest path from $x$ to $t$ that uses $i-1$ or fewer edges, which we've already solved for! So, we just need to take the min over all neighbors $x$ of $v$.

3. How far do we need to go? Answer: at most $i = n - 1$ edges.

Specifically, here is pseudocode for the algorithm. We will use `d[v][i]` to denote the length of the shortest path from $v$ to $t$ that uses $i$ or fewer edges (if it exists) and infinity otherwise ("d" for "distance"). Also, for convenience we will use a base case of $i = 0$ rather than $i = 1$.

**Bellman-Ford pseudocode:**

```
initialize d[v][0] = infinity for v != t.  d[t][i]=0 for all i.
for i=1 to n-1:
   for each v != t:
     d[v][i] = min_{(v,x)∈E} (len(v,x) + d[x][i-1])
For each v, output d[v][n-1].
```

Try it on the above graph!

We already argued for correctness of the algorithm. What about running time? The min operation takes time proportional to the out-degree of $v$. So, the inner for-loop takes time proportional to the sum of the out-degrees of all the nodes, which is $O(m)$. Therefore, the total time is $O(mn)$.

So far we have only calculated the *lengths* of the shortest paths; how can we reconstruct the paths themselves? One easy way is (as usual for DP) to work backwards: if you're at vertex $v$ at distance $d[v]$ from $t$, move to the neighbor $x$ such that $d[v] = d[x] + len(v, x)$. This allows us to reconstruct the path in time $O(m + n)$ which is just a low-order term in the overall running time.

# 1 All-pairs Shortest Paths

Say we want to compute the length of the shortest path between *every* pair of vertices. This is called the **all-pairs** shortest path problem. If we use Bellman-Ford for all $n$ possible destinations $t$, this would take time $O(mn^2)$. We will now see two alternative Dynamic-Programming algorithms for this problem: the first uses the matrix representation of graphs and runs in time $O(n^3 \log n)$; the second, called the *Floyd-Warshall* algorithm uses a different way of breaking into subproblems and runs in time $O(n^3)$.

## 1.1 All-pairs Shortest Paths via Matrix Products

Given a weighted graph $G$, define the matrix $A = A(G)$ as follows:

- $A[i, i] = 0$ for all $i$.
- If there is an edge from $i$ to $j$, then $A[i, j] = len(i, j)$.
- Otherwise, $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from $i$ to $j$ using 1 or fewer edges. Now, following the basic Dynamic Programming idea, can we use this to produce a new matrix $B$ where $B[i, j]$ is the length of the shortest path from $i$ to $j$ using 2 or fewer edges?

Answer: yes. $B[i, j] = \min_k(A[i, k] + A[k, j])$. Think about why this is true!

I.e., what we want to do is compute a matrix product $B = A \times A$ except we change "*" to "+" and we change "+" to "min" in the definition. In other words, instead of computing the sum of products, we compute the min of sums.

What if we now want to get the shortest paths that use 4 or fewer edges? To do this, we just need to compute $C = B \times B$ (using our new definition of matrix product). I.e., to get from $i$ to $j$ using 4 or fewer edges, we need to go from $i$ to some intermediate node $k$ using 2 or fewer edges, and then from $k$ to $j$ using 2 or fewer edges.

So, to solve for all-pairs shortest paths we just need to keep squaring $O(\log n)$ times. Each matrix multiplication takes time $O(n^3)$ so the overall running time is $O(n^3 \log n)$.

## 1.2 All-pairs shortest paths via Floyd-Warshall

Here is an algorithm that shaves off the $O(\log n)$ and runs in time $O(n^3)$. The idea is that instead of increasing the number of edges in the path, we'll increase the set of vertices we allow as intermediate nodes in the path. In other words, starting from the same base case (the shortest path that uses no intermediate nodes), we'll then go on to considering the shortest path that's allowed to use node 1 as an intermediate node, the shortest path that's allowed to use $\{1, 2\}$ as intermediate nodes, and so on.

```
// After each iteration of the outside loop, A[i][j] = length of the
// shortest i->j path that's allowed to use vertices in the set 1..k
for k = 1 to n do:
  for each i,j do:
    A[i][j] = min( A[i][j], (A[i][k] + A[k][j]);
```

I.e., you either go through node $k$ or you don't. The total time for this algorithm is $O(n^3)$. What's amazing here is how compact and simple the code is!