Lecture 11

Dynamic Programming II

11.1 Overview

In this lecture we continue our discussion of dynamic programming, focusing on using it for a variety of path-finding problems in graphs. Topics in this lecture include:

- The Bellman-Ford algorithm for single-source (or single-sink) shortest paths.
- Matrix-product algorithms for all-pairs shortest paths.
- The Floyd-Warshall algorithm for all-pairs shortest paths.
- Dynamic programming for TSP.

11.2 Introduction

As a reminder of basic terminology: a graph is a set of *nodes* or *vertices*, with edges between some of the nodes. We will use V to denote the set of vertices and E to denote the set of edges. If there is an edge between two vertices, we call them *neighbors*. The *degree* of a vertex is the number of neighbors it has. Unless otherwise specified, we will not allow self-loops or multi-edges (multiple edges between the same pair of nodes). As is standard with discussing graphs, we will use $n = |V|$, and $m = |E|$, and we will let $V = \{1, \ldots, n\}$.

The above describes an *undirected* graph. In a *directed* graph, each edge now has a direction (and as we said earlier, we will sometimes call the edges in a directed graph *arcs*). For each node, we can now talk about out-neighbors (and out-degree) and in-neighbors (and in-degree). In a directed graph you may have both an edge from u to v and an edge from v to u.

We will be especially interested here in *weighted* graphs where edges have weights (which we will also call *costs* or *lengths*). For an edge (u, v) in our graph, let's use $len(u, v)$ to denote its weight. The basic shortest-path problem is as follows:

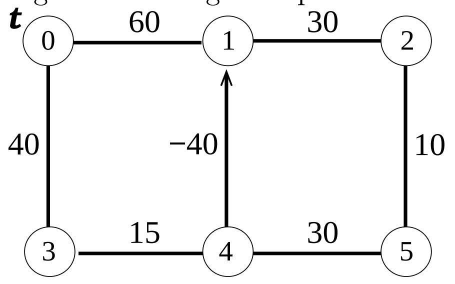
Definition 11.1 *Given a weighted, directed graph G, a start node s and a destination node t, the* **s-t shortest path** *problem is to output the shortest path from s to t. The* **single-source** *shortest* *path problem is to find shortest paths from s to every node in G. The (algorithmically equivalent)* **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 in our drawings, if there is an edge of length ℓ from u to v and also an edge of length ℓ from v to u, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

11.3 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. We will assume the graph is represented as an *adjacency list*: an array of size n where entry v is the list of arcs exiting from node v (so, given a node, we can find all its neighbors in time proportional to the number of neighbors).

Let us develop the algorithm using the following example:



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 ∞ 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 ∞.

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] = $\min_{(v,x)\in E}$ (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.

11.4 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 $O(n^3)$.

11.4.1 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 ($i \neq j$ and there is no edge from i to j), $A[i, j] = \infty$.

I.e., $A[i, j]$ is the length of the shortest path from i to j using 1 or fewer edges.[1] 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.

[1]There are multiple ways to define an adjacency matrix for weighted graphs — e.g., what $A[i, i]$ should be and what $A[i, j]$ should be if there is no edge from i to j. The right definition will typically depend on the problem you want to solve.

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 $O(n^3 \log n)$.

11.4.2 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!

11.5 TSP

The NP-hard *Traveling Salesman Problem* (TSP) asks to find the shortest route that visits *all* vertices in the graph. To be precise, the TSP is the shortest tour that visits all vertices and returns back to the start.[2] Since the problem is NP-hard, we don't expect that Dynamic Programming will give us a polynomial-time algorithm, but perhaps it can still help.

Specifically, the naive algorithm for the TSP is just to run brute-force over all $n!$ permutations of the n vertices and to compute the cost of each, choosing the shortest. (We can reduce this to $(n-1)!$ permutations by always using the same start vertex, but we still pay $\Theta(n)$ to compute the cost of each permutation, so the overall running time is $O(n!)$.) We're going to use Dynamic Programming to reduce this to "only" $O(n^2 2^n)$.

Any ideas? As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to add in some hooks to recover the path. Also, let's work with the shortest-path metric where we've already computed all-pairs-shortest paths (so we can view our graph as a complete graph with weights between any two vertices representing the shortest path between them). This is conveninent since it means a solution is really just a permutation. Finally, let's fix some start vertex s.

[2]Note that under this definition, it doesn't matter which vertex we select as the start. The *Traveling Salesman Path Problem* is the same thing but does not require returning to the start. Both problems are NP-hard.

Now, here is one fact we can use. Suppose someone told you what the initial part of the solution should look like and we want to use this to figure out the rest. Then really all we need to know about it for the purpose of completing it into a tour is the *set* of vertices visited in this initial segment and the *last* vertex t visited in the set. We don't really need the whole ordering of the initial segment. This means there are "only" $n2^n$ subproblems (one for every set of vertices and ending vertex t in the set). Furthermore, we can compute the optimal solution to a subproblem in time $O(n)$ given solutions to smaller subproblems (just look at all possible vertices t' in the set we could have been at right before going to t and take the one that minimizes the cost so far (stored in our lookup table) plus the distance from t' to t).

Here is a top-down way of thinking about it: if we were writing a recursive piece of code to solve this problem, then we would have a bit-vector saying which vertices have been visited so far and a variable saying where we are now, and then we would mark the current location as visited and recursively call on all possible (i.e., not yet visited) next places we might go to. Naively this would take time $\Omega(n!)$. However, by storing the results of our computations we can use the fact that there are "only" $n2^n$ possible (bit-vector, current-location) pairs, and so only that many *different* recursive calls made. For each recursive call we do $O(n)$ work inside the call, for a total of $O(n^2 2^n)$ time.[3]

The last thing is we just need to recompute the paths, but this is easy to do from the computations stored in the same way as we did for shortest paths.

[3]For more, see `http://xkcd.com/399/`