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.
- Algorithms for all-pairs shortest paths, including Floyd-Warshall and Johnson.
- Dynamic programming for the Travelling Salesperson Problem (TSP).

1 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 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.

2 Dijkstra's Algorithm

Let's recall Dijkstra's algorithm: given a directed graph $G = (V, E)$ with positive edge weights, the single-source shortest path problem can be solved using Dijkstra's algorithm (you've seen this in 15-210) in time $O(m \log n)$. This uses a standard heap data-structure; using a fancier data structure called Fibonacci heaps, one can implement Dijkstra's algorithm in $O(m + n \log n)$ time. This will be a benchmark to keep in mind.

3 The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-source) shortest path problem. We will assume the graph is represented as a reversed *adjacency list*: an array P of size n where entry v is the list of vertices from which there is an edge to v. (Given a node v, we can find all the other nodes u such that there is an edge from u to v in time proportional to the number of such neighbors).

How can we use Dyanamic Programming to find the shortest path from s to all other nodes? First of all, as usual for Dynamic Programming, let's focus first on computing the *lengths* of the shortest paths. Later we'll see how to store a little extra information to be able reconstruct the paths themselves. The idea for the algorithm is to keep track of the shortest path from s to v using paths consisting of k or fewer edges.

1. For each node v, find the length of the shortest path from s that uses at zero edges, or write down ∞ if there is no such path.

   This is easy: if $v = s$ this is 0, otherwise it's ∞.

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

   Answer: the shortest path from s to v (where $v \neq s$) that uses k or fewer edges will first go to some neighbor x of v using $k-1$ or fewer edges, then follow the edge (x, v) to get to v. The first part of this path is already known for all x. So, we just need to take the min over all neighbors x of v. To update the value for s, we do the same thing, except we also have to consider its distance using $k-1$ edges, because the path might not have a predecessor.

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

So we can set this up as a recurrence[1] Let $D(v, k)$ be the minimum length path from s to v using k or fewer edges.

$$D(v,k) = \begin{cases} 0 & \text{if } k = 0 \text{ and } v = s \\ \infty & \text{if } k = 0 \text{ and } v \neq s \\ \min\{D(v,k-1), \min\limits_{x \in P(v)} D(x,k-1) + len(x,v)\} & \text{otherwise} \end{cases}$$

The final answer we're looking for is $D(v, n-1)$. We could compute this by memoizing the above recurrence. Alternatively we could do it bottom-up by viewing with $D[v][k]$ as a two dimensional array of distances. The algorithm then becomes:

Bottom-Up Bellman-Ford:

initialize: $D[v][0] \leftarrow$if $v = s$ then 0 else ∞
for $k = 1$ to $n-1$:
  for each $v \in V$:
    $D[v][k] \leftarrow \min\{D[v][k-1], \min\limits_{u \in P(v)} D[u][k-1] + len(u,v)\}$
For each v, output $D[v][n-1]$.

[1]The way the recurrence is written does not exactly mimic the description above. Here we consider the option $D(v, k-1)$ for all nodes v not just s. This removes extra cases in the recurrence and does not change the correctness.

We already argued for correctness of the algorithm. What about running time? The min operation takes time proportional to the in-degree of v. So, the inner for-loop takes time proportional to the sum of the in-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 s, move to the neighbor x such that $d[v] = d[x] + len(x, v)$. This allows us to reconstruct the path in time $O(m + n)$ which is just a low-order term in the overall running time.

Alternatively, we could instrument the algorithm with an additional array of parent pointers. Whenever we compute a new shorter distance from s to v we update v's parent to be the x which caused that to happen. At the end these pointers are a representation of the entire shortest path tree from s to all other vertices.

The algorithm accommodates negative edges. And it can be used to detect if a graph has a negative cycle reachable from s. If there is no such negative cycle, then the shortest path from s to v for all v uses at most $n - 1$ edges. (If it uses more than that it must use a vertex twice and thus involve a cycle.) Thus if we were to make one additonal pass with $k = n$, none of the distances would change. So if we make an additional pass and some of the distances change, this means that there IS a negative cycle reachable from s. The cycle can be found via the use of the parent pointers mentioned above.

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 three 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 $O(n^3)$. The third allows us to adapt Dijkstra's SSSP algorithm to the APSP problem, even on graphs with negative edge lengths.

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.[2] 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!

[2]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.

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

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!

4.3 Adapting Dijkstra's Algorithm for All-pairs

If all the edge lengths in the graph are non-negative, then we can run Dijkstra's algorithm n times, once for each starting vertex. This gives all-pairs shortest paths, and the running time is $O(n(n+m)\log n)$. This is better than Floyd-Warshall on graphs that are sparse. In fact as long as the number of edges is $O(n^2/\log n)$ it equals or beats $O(n^3)$.

On the other hand, it does not handle graphs with negative edge lengths. This can be remedied by a trick discovered by Don Johnson. The idea is to somehow adjust the lengths of all the edges so that (1) they are all non-negative and (2) the shortest paths are the same in the original and in the modified graph.

We're going to compute a real-valued potential on each vertex. The potential of vertex v is denoted Φ_v. If we have an edge (u, v) in the graph of length ℓ, then its length in the modified graph is $\ell' = \ell + (\Phi_u - \Phi_v)$.

Consider a path P of length L from u to v in the original graph. When you sum the modified lengths of the edges on that path it's easy to see that the potential values all telescope and you're left with just the starting minus the ending potential. In other words if L' is the length of the path in the modifed graph, we have:

$$L' = L + \Phi_u - \Phi_v.$$

This is true for any path from u to v. The potential difference is a constant independent of the path. Thus, the shortest path from u to v in the original graph and in the modified graph are exactly the same.

How do we find such a magic set of potentials? Here's how. Create a new vertex called 0, and add an edge of length 0 from vertex 0 to every other vertex. Now find the shortest path from vertex 0 to every other vertex using the Bellman-Ford algorithm. (If there is a negative cycle, this algorithm will find it.) Define Φ_v to be the length of the shortest path from vertex 0 to v.

Now consider an edge (u, v) of length ℓ. We know that

$$\Phi_u + \ell \geq \Phi_v$$

(If this were not true then there would be a shorter path than Φ_v from vertex 0 to v by going through u.)

It follows that:

$$\ell + \Phi_u - \Phi_v \geq 0$$

This is precisely the definition of ℓ'. Which proves that the modified edge lengths are non-negative. The running time of the Bellman-Ford part is $O(nm)$. And the running time of Dijkstra's algorithm is $O(n(n+m)\log n)$. Assuming the graph is connected and that $m \geq n$ this simplifies to $O(nm \log n)$.[3]

5 TSP

The NP-hard *Traveling Salesperson 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.[4] 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 $(n-1)!$ as follows. First of all without loss of generality, we can assume some vertex x is the start vertex. This reduced the number of permutations to try to $(n-1)!$. Secondly, as we generate the permutation, we can keep track of the cost of the permutation up to this point. So there's no need to pay the $O(n)$ cost for each permutation to add up the costs.) We're going to use Dynamic Programming to reduce this to $O(n^2 2^n)$. This is still brute force but it's not as brutish as the naive algorithm.

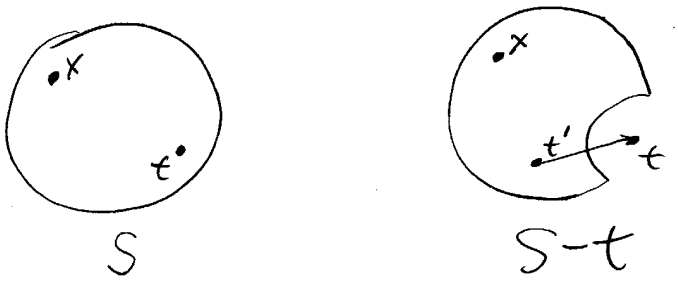
As usual, let's first just worry about computing the *cost* of the optimal solution, and then we'll later be able to add in some hooks to recover the path. Also, let's work with the shortest-path metric where we've already computed all-pairs-shortest paths (using, say, Floyd-Warshall) so we can view 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 call the start vertex x.

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

[3] Using Fibonacci heaps, Dijkstra's algorithm runs in $O(m + n \log n)$. This gives an all-pairs algorithm whose running time is $O(nm + n^2 \log n)$. So even on dense graphs this is no worse asymptotically than Floyd-Warshall.

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

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).



Let's set this up as a recurrence. Let t be a vertex that is different from the start vertex x. Let S be a set of vertices containing x and t. Define $C(S,t)$ as follows:

> $C(S,t)$ = The minimum cost path starting from x and ending at t and hitting all vertices in S along the way.

Here's a recurrence for $C(S,t)$:

$$C(S,t) = \begin{cases} \text{len}(x,t) & \text{if } S = \{x,t\} \\ \min_{t' \in S, t' \neq t, t' \neq x} C(S-t,t') + \text{len}(t',t) & \text{otherwise} \end{cases}$$

The parameter space for $C(S,t)$ is 2^{n-1} (the number of subsets S considered) times n (the number of choices for t). For each recursive call we do $O(n)$ work inside the call, for a total of $O(n^2 2^n)$ time.[5]

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.

This technique is sometimes called "Subset DP". These ideas apply in many cases to reduce a factorial running to time to a regular exponential running time. A very nice way to implement these kinds of algorithms in practice is to represent the set S as an integer, where a 1 bit in position b means the presence of b in the set. This facilitates memoization, because the set is now just an integer that can quickly used to index an array or be looked up in a hash table.

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