13.1 Introduction

Today we're going to talk about algorithms for computing shortest paths in graphs. If all edges have length 1, we saw last class that a BFS computes shortest paths in time $O(m+n)$. But what if edges have arbitrary edge lengths? Are there still fast algorithms? We'll see a few different algorithms for versions of this problem.

Some technicalities. For today, we'll only be concerned with *single-source* shortest paths, i.e. we will try to design algorithms that when given a graph G and a node v, computes the shortest paths from v to all other nodes. How do we represent this? It's not hard to see that if w is on the shortest path from v to u, then the shortest path from v to w must be a prefix of the shortest path from v to u. This means that the set of shortest paths out of v form a tree, which is usually known as the *shortest-path tree*.

This also lets us represent the set of shortest paths compactly. At the completion of the algorithms, we require that every node have two pieces of data: tits distance from v, and its parent $v.parent$ in the shortest-path tree.

I'm generally going to think of the graph as being directed, so edges are directed. If the graph is undirected, we can just turn each edge $\{x, y\}$ into two edge (x, y) and (y, x), which only increases the number of edges by a factor of 2. For an edge (x, y), let $len(x, y)$ denote its length.

13.2 Relaxation

One thing which many shortest-path algorithms have in common is the notion of *relaxing* an edge. Since we require that in the end every node knows its distance from v, let's assume that each node u always has an upper bound $\hat{d}(u)$ on its real distance from v (in reality we would store this in some field, so it would be $u.d$, but let's stick with math notation). We initialize $\hat{d}(v)$ to 0, and for every other node x we set $\hat{d}(x) = \infty$.

Relaxing edge (x, y) simply checks if we can decrease $\hat{d}(y)$ by using $\hat{d}(x)$ and $len(x, y)$. We test whether $\hat{d}(x) + len(x, y) < \hat{d}(y)$, and if so we update $\hat{d}(y)$ to $\hat{d}(x) + len(x, y)$ and update $y.parent$ to x. We do this because it means that our best current guess for the shortest path to y goes through x. Many shortest-path algorithms use edge relaxations – the main question is what order to relax edges in.

13.3 Bellman-Ford

The first algorithm we're going to talk about is Bellman-Ford. The basic idea of Bellman-Ford is extremely simple: iterate over all vertices, relaxing each edge incident on the vertex. Repeat n

times.

The pseudocode is the following:

```
for (i = 1 to n) {
  foreach (u ∈ V} {
    foreach edge (u,x), relax(u,x)
  }
}
```

The running time is clearly $O(nm)$ – each of the n iterations of the outer loops causes us to relax every edge once, and relaxing an edge takes $O(1)$ time.

To prove that the algorithm does return shortest paths, let's actually prove something a little stronger. Before we do that, though, there is one subtlety: when we relax an edge, do we use the new distance bounds on later relaxations from the same iterations or not? In other words, when we relax do we compute a new distance estimate $d'(u)$ for all u, and only at the end of the iteration copy the value into $\hat{d}$?

On the one hand, doing this cannot help our estimate – doing a real relaxation can be much better in some cases, and is never worse. On the other hand, freezing the values during a single iteration has a few benefits. First, one reason Bellman-Ford is used quite a bit in practice is that it is highly distributed/parallel – each node can relax all of its own edges in parallel, or even more extremely each edge can be relaxed in parallel. If we're in such a situation, then we don't want these parallel computations to interact with each other by trying to simultaneously write to shared memory or anything like that. So instead we freeze the distance estimates, do the parallel relaxations, and then update all of the estimates by letting $\hat{d}(u)$ be the minimum of the estimates computed by relaxations of edges going into u.

For our purposes, though, freezing the values $\hat{d}$ inside of an iteration makes the algorithm a little easier to reason about and also gives it a couple nice properties. The following is the most important:

Theorem 13.3.1 *After k iterations, every node knows the shortest path from v that uses at most k edges and the distance of this path.*

Proof: We prove this by induction on the number of iterations. After the first iteration, the only nodes whose distance estimate is noninfinite are exactly the nodes with a path of length 1 from v. And clearly when we relax those edges, the distance estimates become exactly the length of those edges.

So now we have to prove the inductive case. Consider iteration k, and suppose that the theorem is true for all $k' < k$. Let $u \in V$ be an arbitrary node for which there exists a path from v to u with at most k edges. Let w be the node immediately before u on this path. So the shortest path with at most k edges from v to u consists of a shortest path from v to w with at most $k-1$ edges, and then the edge (w, u). By induction, before the kth iteration starts the node w knows its distance from v along this path. So when we relax the edge (w, u) we get a distance estimate or u which exactly corresponds to this path. Since this is the shortest possible path of this form, this is the estimate which we will end up with at the end of the iteration. ■

Note that this proof crucially used the fact that distance estimates are frozen. Otherwise, we would

only be able to prove that after k iterations, every node u knows a path from v that is *no longer* than the shortest path from v to u using at most k edges.

In any case, this theorem implies that after n iterations, every nodes knows the actual shortest path from v.

13.3.1 Dynamic Programming

Another way of looking at Bellman-Ford is through dynamic programming. We'll have our table M like usual, and we'll say that $M[u, k]$ contains the length of the shortest path from v to u with at most k edges. Then we get the recurrence

$$M[u, k] = \min_{w:(w,u)\in E} (M[w, k-1] + len(w, u))$$

This gives a dynamic programming algorithm that takes time $O(mn)$, and is equivalent to Bellman-Ford.

13.3.2 Negative Weights and Cycles

In some settings it is very natural to have negative edge lengths. This causes a few issues. First, what happens if there's a cycle where the sum of edge lengths along the cycle are negative? Then shortest paths aren't even really defined – you can always enter the cycle, go around it as many times as you want to decrease your length as much as you want, and then go to the destination. So the length of the shortest path between any two nodes is always $-\infty$. If you restrict your paths to simple paths (i.e. don't allow yourself to loop) then in fact the problem becomes NP-hard, which as we'll see later means that we don't expect a fast algorithm to exist.

So there are problems if there are negative cycles. What will Bellman-Ford do? Well, if there are no negative cycles then after $n-1$ iterations we will have found shortest paths. But if there are negative cycles, then after $n-1$ iterations further relaxations would allow shorter paths. So when we finish Bellman-Ford, we can do one more round of relaxations, and if anything changes we know there is a negative cycle. So Bellman-Ford can be used for negative-cycle detection as well.

So let's suppose there are no negative cycles. Then everything we did for Bellman-Ford still works! So Bellman-Ford can be used in general when there are negative weights – it can find a negative cycle of one exists, and if one does not exist then it works correctly even if some weights are negative.

13.4 Dijkstra's Algorithm

Probably the most famous shortest-path algorithm is Dijkstra's algorithms. As we'll see, it is faster than Bellman-Ford, but has the drawback that is does not work if there are edges of negative length (even if there are no negative-weight cycles).

Dijkstra's algorithm can be thought of as a greedy algorithm, which is a paradigm that we'll talk more about later. The algorithm itself is pretty simple. For every node we maintain a distance guess $\hat{d}(u)$, like in Bellman-Ford. We initialize $\hat{d}(v)$ to 0, and for all other u we initialize $\hat{d}(u)$ to

∞. We will also maintain a tree T which starts out empty (T will be the shortest-path tree in the end).

We then do the following: until T contains all of the nodes, we choose the node u with smallest $\hat{d}(u)$ and add it to the tree (using whichever edge cause the distance estimate). We then relax all edges between u and non-tree nodes.

Before we analyze the running time, let's prove correctness. We will do this inductively.

Theorem 13.4.1 *Throughout the algorithm, T is a shortest-path tree from v to the nodes in T, and for every node u in T their distance estimate $\hat{d}(u)$ is equal to their actual shortest path from v.*

Proof: In the first step, v is added to the tree with $\hat{d}(v) = 0$, so by definition the theorem is true at that time.

To prove the inductive step, suppose that it is true at some point and we have tree T. Let u be the next vertex that we add, and suppose we add it using edge (w, u). By induction we know that $\hat{d}(w) = d(w)$, so when we add u we have $\hat{d}(u) = d(w) + len(w, u)$.

Suppose that this path to u (through the tree to w, then to u) is not the shortest-path. So there is some other path P which has length less than $\hat{d}(u)$. First, note that the last node w' on this path cannot already be in T: if it were, then by induction $\hat{d}(w') = d(w')$ and we would have relaxed the edge (w', u), so $\hat{d}(u)$ would equal $\hat{d}(w') + len(w', u)$. This would be a contradiction, since we know that $\hat{d}(u) = d(w) + len(w, u) > d(w') + len(w', u)$.

So suppose that w' is not in T. Then let x be the *first* node in P not in T (this might be w', but might not be), and let y be the node just before x (so $y \in T$). Then when we added y to T we relaxed the edge (y, x), so $\hat{d}(x)$ is at most $\hat{d}(y) + len(y, x) = d(y) + len(y, x)$. Since we're assuming that x is on the real shortest path to u and that this path goes through y, so in fact we know that $\hat{d}(x) = d(x) < d(u) \leq \hat{d}(u)$. This gives a contradiction, since the algorithm would not pick u to be the next node added but would instead pick x. ■

It's a good exercise to see where this proof used the fact that all weights are nonnegative. These proofs are also done in more detail in the book.

13.4.1 Running Time

Interestingly, the running time of this algorithm depends heavily on the data structures that we use (which is I think the first time that we've seen this happen).

What are the operations done by this algorithm? We have to be able to select the node with minimum $\hat{d}$, and we have to do this n times. Every edge also gets relaxed once, so we have to (possibly) decrease values of $\hat{d}$ a total of m times.

If we simply keep $\hat{d}(u)$ with the node u, and keep the adjacency list representation in an arbitrary order, the first kind of operation takes time $\Theta(n)$ and relaxing the edges takes time $O(1)$ per relaxation, so the total running time is $O(n^2 + m) = O(n^2)$.

But what kind of data structure lets us quickly find the minimum of a set of values, and also decrease values in the set? A heap! We just need a heap that let's us do Insert, Extract-Min, and Decrease-Key quickly. If we use a binary heap, we pay $O(\log n)$ for all of those operations. Thus the

total running time becomes $O(m \log n + n \log n) = O(m \log n)$ (assuming the graph is connected).

There's no point in moving to a binomial heap, since we don't care in this context about Meld. But there are fancier heaps that can decrease the running time further. The famous example (and currently the best known in this context) is a Fibonacci heap. Fibonacci heaps only take $O(1)$ time (amortized) for Decrease-Key and Insert, and Extract-Min only takes time $O(\log n)$ (amortized). So the total running time if we use Fibonacci heaps is only $O(m + n \log n)$!

Note that the amortized guarantee is enough for us here, since we're only concerned about the total running time of a sequence of operations (exactly the context where amortization works).