Lecture 23

Floyd-Warshall all-pairs shortest paths and minimum spanning tree algorithms of Prim and Kruskal

Wed Nov 16, 2022
Reading: Weiss 10.3.4, 9.5

A pdf version of these notes is available.

All-pairs shortest paths

Last time, we ended with a discussion on the all-pairs shortest paths problem. Given a directed graph with weighted edges, compute the shortest path distance between every pair of vertices by storing them in a 2D array dist[][] . Entry dist[i][j] should be the distance between i and j.

The solution to getting a fast algorithm even with negative weight edges (but no negative weight cycles) lies in dynamic programming. Let \( n = |V| \), and let's arbitrarily number the vertices 0 through \( n - 1 \). Now, consider two vertices \( i \) and \( j \) along with some integer \( 0 \leq k \leq n - 1 \). An internal vertex of a path is any vertex except the first \( (i) \) or the last \( (j) \) . Let's specifically consider the shortest path from \( i \) to \( j \) that has for internal vertices only those numbered 0 through \( k \), and let \( D_{i,j,k} \) denote the length of that path. \( D_{i,j,n-1} \) is the shortest path distance from \( i \) to \( j \).

Alright, so this path does one of two things. It either uses vertex \( k \) itself as an internal
vertex, or it doesn't. If it doesn't, then it has internal vertices numbered 0 through \( k - 1 \), implying \( D_{i,j,k} = D_{i,j,k-1} \). If it does have internal vertex \( k \), then the subpath from \( i \) to \( k \) has internal vertices numbered 0 through \( k - 1 \), and so does the subpath from \( k \) to \( j \). In this case, \( D_{i,j,k} = D_{i,k,k-1} + D_{k,j,k-1} \). We don't actually know in advance which case is correct, but we do know the one with smallest total is the real value for \( D_{i,j,k} \). In other words.

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

We can compute all the \( D_{i,j,k} \) values using dynamic programming by storing each value in a 3D array indexed by \( i \), \( j \), and \( k \). Each subproblem depends upon those with smaller \( k \) index, so we can use an outer for loop going from \( k = 0 \) to \( k = n - 1 \), and choose our \( i \) and \( j \) in arbitrary order. Each entry takes \( O(1) \) time to compute, so the total running time will be \( O(n^3) = O(|V|^3) \). Great!

OK, but let's clean some things up before I show you the code. First, the 3D array uses \( O(|V|^3) \) space, which is a lot more than we really need (\( O(|V|^2) \)). We really only need the values for \( k = 1 \). In fact, it's not actually that important that we're explicitly acknowledging the internal vertices have labels through \( k \). We can take a sort of Bellman-Ford approach and run the algorithm in phases where the \( k \)th phase ends with all distances being at most the shortest path distance with internal vertices labeled 0 through \( k \). The resulting algorithm is often called the Floyd-Warshall algorithm for all-pairs shortest paths.

```java
// Method takes an adjacency matrix a storing edge weights.
// We'll fill in 2D array d[][] where d[i] is the single source
// shortest-paths distance array for source i.
public static void allPairs(int[][] a, int[][] d) {
    int n = a.length;

    // Initialize d with D[i][j][-1] (no internal vertices)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            d[i][j] = a[i][j];
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (d[i][k] + d[k][j] < d[i][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                }
            }
        }
    }
}
```

**Minimum spanning tree**

Let's discuss one more graph problem. Here, you're given a connected *undirected* graph
A spanning tree is a connected undirected subgraph that contains every vertex (it is spanning) and is acyclic (it is a tree). A minimum spanning tree (MST) is a spanning tree of minimum total weight. Every spanning tree has exactly $|V| - 1$ edges, so every spanning tree is a minimum spanning tree if edges are unweighted. But only some spanning trees are minimum if the edges have arbitrary weights. Note that weights can be negative, but it doesn’t affect the algorithms at all.

All basic minimum spanning tree algorithms follow the same strategy: we start with an empty set of edges and proceed in a series of rounds. In each round, we add one or more edges to our collection chosen so we can guarantee all edges we have selected belong to some minimum spanning tree. The hard part is knowing which edges are safe to add in each round.

Our collection of edges forms a subset of the minimum spanning tree edges, so they form a spanning forest consisting of between 1 and $|V|$ connected components, each of which is a tree. Say an edge $e$ leaves a particular component if it has exactly one endpoint in that component. It turns out we can pick any of the components and then take the cheapest edge leaving that component. In fact, we can grab a cheapest edge leaving multiple components at the same time, as long as we don’t create a cycle using our new edges.

To see why this works, let $T$ be any minimum spanning tree containing the edges of component $S$. Let $e$ be a cheapest edge leaving $S$. If $T$ contains $e$, then great, $T$ is a minimum spanning tree containing $e$. Otherwise, there must be a path between the endpoints of $e$ in $T$, and in particular, there must be some other edge $e’$ on that path that leaves $S$ instead. Well, if we remove $e’$ from $T$ and add $e$ in its place, we’ll still have a spanning tree, and it won’t cost any more than when we started, so we’ll have a new minimum spanning tree containing $e$. This sort of exchange argument is used all the time to prove correctness of greedy algorithms.

**Prim’s algorithm**

So now that we know the general strategy, we have to decide which edge(s) we’re going to add in each round. The first algorithm we’ll look at is usually called Prim’s algorithm. For this algorithm, we focus on growing a single component of the spanning forest starting with an arbitrary vertex $s$. Each round, we select exactly one cheapest edge leaving the component to put in our minimum spanning tree.

The trick to making this strategy run quickly is to basically treat it as a BFS, except instead of a queue, we use a priority queue containing the vertices outside of the component where the key is the cheapest edge from the component to that vertex. Every time we add a new edge going to some vertex $v$, we look at all the edges incident to $v$ and leaving the component, updating the members of the priority queue appropriately.

```java
ArrayList<Edge> prim() {
    PriorityQueue<Vertex> pq = new PriorityQueue<Vertex>();
    List<Edge> mst = new ArrayList<Edge>();
    for each Vertex v {
        v.key = INFINITY;
        pq.insert(v);
        v.incoming = null; // Need to remember cheapest edge to v.
        v.known = false; // Need to track if v is in the main component.
```
This code looks a lot like Dijkstra's algorithm, and it actually has the same running time! If we use a binary heap for our priority queue, it runs in $O(|E| \log |V|)$ time, and if we use a Fibonacci heap, it runs in $O(|V| \log |V| + |E|)$ time.

**Kruskal's algorithm**

The next algorithm we're going to look at can't be made to go quite as fast in the worst-case, but it's important to the theory of computing minimum spanning trees and other greedy algorithms. This algorithm is usually called Kruskal's algorithm.

The big idea is to simply consider each edge in increasing order of cost. If the edge can be added to our collection without creating a cycle, we add it. Otherwise, we don't. The argument that it works is a short proof by induction. Suppose we're considering some edge $e$. Inductively, all the edges in the forest so far are a subset of those in one or more minimum spanning trees. If adding $e$ would create a cycle, then $e$ obviously doesn't belong to any of those trees. Otherwise, $e$ is a cheapest edge leaving both of its components, because a cheaper edge would have been added to the forest already.

To implement the algorithm, we need some way to consider the edges in increasing weight order. We could sort them in $O(|E| \log |E|) = O(|E| \log |V|)$ time and then walk down the sorted list. Alternatively, we could put them in a binary heap in $O(|E|)$ time using `buildHeap` and then repeatedly remove the edge of least cost. If we're lucky, we could pull out as few as $|V|$ edges, spending $O(|V| \log |V|)$ time with the queue, although we could still spend $O(|E| \log |V|)$ time in the worst case doing up to $|E|$ binary heap operations.

The other thing we need to figure out is how to take an edge and quickly determine if adding it will create a cycle. It's finally time to use that disjoint sets data structure we talked about a few weeks ago. We'll build a disjoint sets structure over the vertices, and
make sure each set contains exactly those vertices in a single connected component of
the forest. If both endpoints of an edge belong to the same set, then they belong to the
same component and we cannot add the edge. Otherwise, we can add the edge to the
forest and then do a union to declare that both components have been merged into one.
The up to $|E|$ union and find operations take only $O(\alpha(|E|)) = O(\alpha(|V|))$ amortized
time each, so knowing what order to consider the edges actually takes longer than figuring
out if we can add them to the minimum spanning tree.

```java
ArrayList<Edge> kruskal() {
    numVertices = number of vertices;
    DisjSets ds = new DisjSets(numVertices);
    PriorityQueue<Vertex> pq = new PriorityQueue<Edge>();
    List<Edge> mst = new ArrayList<Edge>();
    
    for each Edge e {
        pq.insert(e); // Keyed by edge weight.
    }
    
    while (mst.size() < numVertices - 1) {
        Edge e = pq.deleteMin();
        SetType uset = ds.find(e.getu()); // e = uv
        SetType vset = ds.find(e.getv());
        
        if (uset != vset) {
            mst.add(e);
            ds.union(uset, vset);
        }
    }
    
    return mst;
}
```