Lecture 13

Graph Algorithms II

13.1 Overview

In this lecture we begin with one more algorithm for the shortest path problem, *Dijkstra's algorithm*. We then will see how the basic approach of this algorithm can be used to solve other problems including finding *maximum bottleneck paths* and the *minimum spanning tree* (MST) problem. We will then expand on the minimum spanning tree problem, giving one more algorithm, *Kruskal's algorithm*, which to implement efficiently requires an good data structure for something called the *union-find problem*. Topics in this lecture include:

- Dijkstra's algorithm for shortest paths when no edges have negative weight.
- The Maximum Bottleneck Path problem.
- Minimum Spanning Trees: Prim's algorithm and Kruskal's algorithm.

13.2 Shortest paths revisited: Dijkstra's algorithm

Recall the *single-source* shortest path problem: given a graph G, and a start node s, we want to find the shortest path from s to all other nodes in G. These shortest paths can all be described by a tree called the *shortest path tree* from start node s.

Definition 13.1 *A* **Shortest Path Tree** *in G from start node s is a tree (directed outward from s if G is a directed graph) such that the shortest path in G from s to any destination vertex t is the path from s to t in the tree.*

Why must such a tree exist? The reason is that if the shortest path from s to t goes through some intermediate vertex v, then it must use a shortest path from s to v. Thus, every vertex $t \neq s$ can be assigned a "parent", namely the second-to-last vertex in this path (if there are multiple equally-short paths, pick one arbitrarily), creating a tree. In fact, the Bellman-Ford Dynamic-Programming algorithm from the last class was based on this "optimal subproblem" property.

The first algorithm for today, *Dijkstra's algorithm*, builds the tree outward from s in a greedy fashion. Dijkstra's algorithm is faster than Bellman-Ford. However, it requires that all edge

lengths be non-negative. *See if you can figure out where the proof of correctness of this algorithm requires non-negativity.*

We will describe the algorithm the way one views it conceptually, rather than the way one would code it up (we will discuss that after proving correctness).

Dijkstra's Algorithm:

Input: Graph G, with each edge e having a length $len(e)$, and a start node s.

Initialize: tree $= \{s\}$, no edges. Label s as having distance 0 to itself.

Invariant: nodes in the tree are labeled with the correct distance to s.

Repeat:

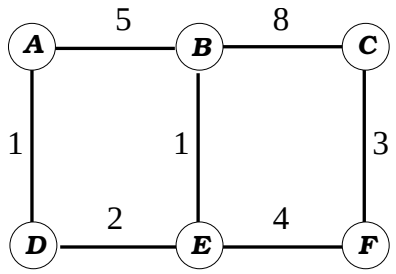
1. For each neighbor x of the tree, compute an (over)-estimate of its distance to s:

$$\text{distance}(x) = \min_{e=(v,x):v\in\text{tree}} [\text{distance}(v) + len(e)] \tag{13.1}$$

   In other words, by our invariant, this is the length of the shortest path to x whose only edge not in the tree is the very last edge.

2. Insert the node x of minimum distance into tree, connecting it via the argmin (the edge e used to get distance(x) in the expression (13.1)).

Let us run the algorithm on the following example starting from vertex A:



Theorem 13.1 *Dijkstra's algorithm correctly produces a shortest path tree from start node s. Specifically, even if some of distances in step 1 are too large, the* minimum *one is correct.*

Proof: Say x is the neighbor of the tree of smallest distance(x). Let P denote the *true* shortest path from s to x, choosing the one with the fewest non-tree edges if there are ties. What we need to argue is that the last edge in P must come directly from the tree. Let's argue this by contradiction. Suppose instead the first non-tree vertex in P is some node $y \neq x$. Then, the length of P must be at least distance(y), and by definition, distance(x) is smaller (or at least as small if there is a tie). This contradicts the definition of P. ■

Did you catch where "non-negativity" came in in the proof? Can you find an example with negative-weight directed edges where Dijkstra's algorithm actually fails?

Running time: To implement this efficiently, rather than recomputing the distances every time in step 1, you simply want to update the ones that actually are affected when a new node is added to the tree in step 2, namely the neighbors of the node added. If you use a heap data structure to store the neighbors of the tree, you can get a running time of $O(m \log n)$. In particular, you can start by giving all nodes a distance of infinity except for the start with a distance of 0, and putting all nodes into a min-heap. Then, repeatedly pull off the minimum and update its neighbors, tentatively assigning parents whenever the distance of some node is lowered. It takes linear time to initialize the heap, and then we perform m updates at a cost of $O(\log n)$ each for a total time of $O(m \log n)$.

If you use something called a "Fibonacci heap" (that we're not going to talk about) you can actually get the running time down to $O(m+n \log n)$. The key point about the Fibonacci heap is that while it takes $O(\log n)$ time to remove the minimum element just like a standard heap (an operation we perform n times), it takes only amortized $O(1)$ time to decrease the value of any given key (an operation we perform m times).

13.3 Maximum-bottleneck path

Here is another problem you can solve with this type of algorithm, called the "maximum bottleneck path" problem. Imagine the edge weights represent capacities of the edges ("widths" rather than "lengths") and you want the path between two nodes whose minimum width is largest. How could you modify Dijkstra's algorithm to solve this?

To be clear, define the *width* of a path to be the minimum width of any edge on the path, and for a vertex v, define widthto(v) to be the width of the widest path from s to v (say that widthto(s) $= \infty$). To modify Dijkstra's algorithm, we just need to change the update rule to:

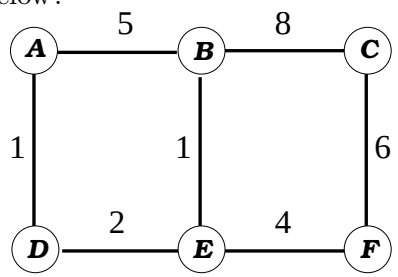
$$\text{widthto}(x) = \max_{e=(v,x):v \in \text{tree}} [\min(\text{widthto}(v), \text{width(e)})]$$

and now put the node x of *maximum* "widthto" into tree, connecting it via the argmax. We'll actually use this later in the course.

13.4 Minimum Spanning Trees

A **spanning tree** of a graph is a tree that touches all the vertices (so, it only makes sense in a connected graph). A **minimum spanning tree** (MST) is a spanning tree whose sum of edge lengths is as short as possible (there may be more than one). We will sometimes call the sum of edge lengths in a tree the *size* of the tree. For instance, imagine you are setting up a communication network among a set of sites and you want to use the least amount of wire possible. *Note:* our definition is only for *undirected* graphs.

What is the MST in the graph below?



13.4.1 Prim's algorithm

Prim's algorithm is an MST algorithm that works much like Dijkstra's algorithm does for shortest path trees. In fact, it's even simpler (though the correctness proof is a bit trickier).

Prim's Algorithm:

1. Pick some arbitrary start node s. Initialize tree $T = \{s\}$.
2. Repeatedly add the shortest edge incident to T (the shortest edge having one vertex in T and one vertex not in T) until the tree spans all the nodes.

So the algorithm is the same as Dijsktra's algorithm, except you don't add distance(v) to the length of the edge when deciding which edge to put in next. For instance, what does Prim's algorithm do on the above graph?

Before proving correctness for the algorithm, we first need a useful fact about spanning trees: if you take any spanning tree and add a new edge to it, this creates a cycle. The reason is that there already was one path between the endpoints (since it's a *spanning* tree), and now there are two. If you then remove any edge in the cycle, you get back a spanning tree (removing one edge from a cycle cannot disconnect a graph).

Theorem 13.2 *Prim's algorithm correctly finds a minimum spanning tree of the given graph.*

Proof: We will prove correctness by induction. Let G be the given graph. Our inductive hypothesis will be that the tree T constructed so far is consistent with (is a subtree of) some minimum spanning tree M of G. This is certainly true at the start. Now, let e be the edge chosen by the algorithm. We need to argue that the new tree, $T \cup \{e\}$ is also consistent with some minimum spanning tree M' of G. If $e \in M$ then we are done ($M' = M$). Else, we argue as follows.

Consider adding e to M. As noted above, this creates a cycle. Since e has one endpoint in T and one outside T, if we trace around this cycle we must eventually get to an edge e' that goes back in to T. We know $len(e') \geq len(e)$ by definition of the algorithm. So, if we add e to M and remove e', we get a new tree M' that is no larger than M was and contains $T \cup \{e\}$, maintaining our induction and proving the theorem. ■

Running time: We can implement this in the same was as Dijkstra's algorithm, getting an $O(m \log n)$ running time if we use a standard heap, or $O(m + n \log n)$ running time if we use a Fibonacci heap. The only difference with Dijkstra's algorithm is that when we store the neighbors of T in a heap, we use priority values equal to the shortest edge connecting them to T (rather than the smallest sum of "edge length plus distance of endpoint to s").

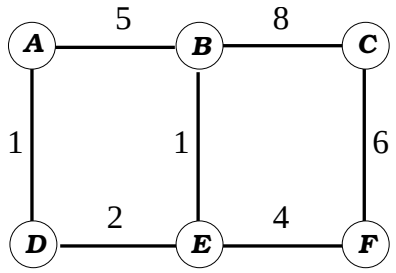
13.4.2 Kruskal's algorithm

Here is another algorithm for finding minimum spanning trees called Kruskal's algorithm. It is also greedy but works in a different way.

Kruskal's Algorithm:

> Sort edges by length and examine them from shortest to longest. Put each edge into the current forest (a forest is just a set of trees) if it doesn't form a cycle with the edges chosen so far.

E.g., let's look at how it behaves in the graph below:



Kruskal's algorithm sorts the edges and then puts them in one at a time so long as they don't form a cycle. So, first the AD and BE edges will be added, then the DE edge, and then the EF edge. The AB edge will be skipped over because it forms a cycle, and finally the CF edge will be added (at that point you can either notice that you have included $n - 1$ edges and therefore are done, or else keep going, skipping over all the remaining edges one at a time).

Theorem 13.3 *Kruskal's algorithm correctly finds a minimum spanning tree of the given graph.*

Proof: We can use a similar argument to the one we used for Prim's algorithm. Let G be the given graph, and let F be the forest we have constructed so far (initially, F consists of n trees of 1 node each, and at each step two trees get merged until finally F is just a single tree at the end). Assume by induction that there exists an MST M of G that is consistent with F, i.e., all edges in F are also in M; this is clearly true at the start when F has no edges. Let e be the next edge added by the algorithm. Our goal is to show that there exists an MST M' of G consistent with $F \cup \{e\}$.

If $e \in M$ then we are done ($M' = M$). Else add e into M, creating a cycle. Since the two endpoints of e were in different trees of F, if you follow around the cycle you must eventually traverse some edge $e' \neq e$ whose endpoints are also in two different trees of F (because you eventually have to get back to the node you started from). Now, both e and e' were eligible to be added into F, which by definition of our algorithm means that $len(e) \leq len(e')$. So, adding e and removing e' from M creates a tree M' that is also a MST and contains $F \cup \{e\}$, as desired. ■

Running time: The first step is sorting the edges by length which takes time $O(m \log m)$. Then, for each edge we need to test if it connects two different components. This seems like it should be a real pain: how can we tell if an edge has both endpoints in the same component? It turns out there's a nice data structure called the *Union-Find* data structure for doing this operation. It is so efficient that it actually will be a low-order cost compared to the sorting step.

We will talk about the union-find problem in the next class, but just as a preview, the *simpler* version of that data structure takes time $O(m+n \log n)$ for our series of operations. This is already good enough for us, since it is low-order compared to the sorting time. There is also a more sophisticated version, however, whose total time is $O(m \lg^* n)$, in fact $O(m\alpha(n))$, where $\alpha(n)$ is the inverse-Ackermann function that grows even more slowly than $\lg^*$.

What is $\lg^*$? $\lg^*(n)$ is the number of times you need to take $\log_2$ until you get down to 1. So,

$$\begin{aligned} \lg^*(2) &= 1 \\ \lg^*(2^2 = 4) &= 2 \\ \lg^*(2^4 = 16) &= 3 \\ \lg^*(2^{16} = 65536) &= 4 \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

I won't define Ackerman, but to get $\alpha(n)$ up to 5, you need n to be at least a stack of 256 2's.