Illinois Institute of Technology
Department of Computer Science

Lectures 20–21: November 12–17, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

1 Minimum Spanning Trees

Consider a connected, undirected graph $G = (V, E)$. For many applications it is useful to find a subgraph $T = (V, E')$ of G, where E' is the smallest collection of edges sufficient to make T a connected graph, i.e. T is a tree. Subgraphs such as T are called *spanning trees* of G. This problem can be made more general by assigning each edge of E a weight and finding the spanning tree whose edges have the smallest combined weight. Such a spanning tree is called a *minimum spanning tree.* It turns out that such trees can be found efficiently. We examine a number of algorithms for doing so.[1]

1.1 Prim's Algorithm

The first algorithm we examine for the minimum spanning tree problem is called Prim's algorithm. In this algorithm, the spanning tree is "grown" from a single initial vertex, called the *source vertex.* Each step of the algorithm involves adding a single edge and vertex to the existing tree. This is done in a greedy manner by adding the cheapest edge from the current tree to some "outside" vertex. This can be implemented efficiently by putting the vertices into a priority queue. The source vertex is initially assigned weight 0 and all other vertices are assigned infinite weight. These weights are an estimate of the distance from the vertex to the current tree. When a vertex u is removed from the queue all its edges are examined. For each outgoing edge which goes to a vertex not already in the tree, the cost of including that edge is compared to the current weight of that vertex. If the examined edge is cheaper, then the distance estimate is revised by calling Decrease-Key to adjust the weight of that vertex. If the vertices each remember the cheapest edge connecting them to the tree, then the edges which should be added to the tree can be easily determined as well.

Now we show that this algorithm produces a minimum spanning tree. As with other greedy algorithms, we do this by looking at the algorithm's first mistake. Let this first mistake occur when we add some vertex v to the tree T using an edge uv which is not in any optimal minimum spanning tree of the graph. Let T_{opt} be an optimal tree which includes the edges in T. There must be a path from u to v in this tree. Since u is inside T and v is not, at some point this path must cross from vertices in T to vertices not in T. Let wx be the first edge leaving the vertices of T along this path. Clearly removing wx from T_{opt} and replacing it with uv creates a spanning tree. Furthermore, since we choose to connect a vertex via the cheapest outgoing edge at each step, the edge wx must cost at least as much as uv. Thus, the new spanning tree we create must cost no more than T_{opt}, making it a minimum spanning tree. However, since $T + uv$ is a subset of this tree, we must not have made a mistake adding uv. Therefore, the algorithm produces a minimum spanning tree.

What is the running time of this algorithm? If we use a Fibonacci heap to implement the priority queue, the vertices can be added to the queue in $O(V)$ time, the vertices can be extracted in a total of $O(V \log V)$ time, and the keys can be decreased in a total of $O(E)$ time. Thus the entire algorithm runs in $O(E + V \log V)$ time.

[1]This problem is discussed in chapter 23 of CLRS, though it is given a slightly different treatment here.

1.2 Kruskal's Algorithm

Kruskal's algorithm is a different greedy approach to finding a minimum spanning tree. Rather than starting from a single source vertex as we do with Prim's algorithm, Kruskal's algorithm grows separate subtrees until they are eventually connected to form a minimum spanning tree. Initially, each vertex is a trivial subtree. At each subsequent step, we add the cheapest edge which connects different subtrees.

First we show that this algorithm produces a minimum spanning tree, again by analyzing the first "mistake" made by the algorithm. Suppose that some set S of edges has already been correctly added before some edge uv not in any minimum spanning tree is added. Let T_{opt} be a minimum spanning tree which uses the edges of S. There must be a path between vertices u and v in T_{opt}. Since u and v were in separate components before that edge was added, there must be some first edge wx not in S along this path. Swap uv for this edge in T_{opt}. Since we choose edges in order of weight and have not chosen wx yet, the resulting spanning tree must also be minimal and therefore the algorithm did not make a mistake.

Now we consider the running time of Kruskal's algorithm. First the algorithm sorts the edges by weight, in $O(E \log E)$ time. In order to keep track of which vertices are already connected, we keep track of the components with a disjoint set data structure. Thus, for initialization we build $|V|$ new sets (calling the MAKE-SET operation on each vertex) in $O(V)$ time. For each edge (u, v) considered in increasing order, we determine if u and v are in the same tree by comparing FIND-SET(u) with FIND-SET(v). If u and v are indeed in different trees, we add the edge (u, v) and UNION the two trees. Since at worst we do 2 FIND-SET operations per edge, this loop takes at most $O(E\alpha(V)) \leq O(E \log V)$ time. Thus the full algorithm takes $O(E \log E)$ time. This is asymptotically more than the running time of Prim's algorithm, but Kruskal's may be more practical for small problem instances since it does not require a Fibonacci heap, which introduces some very large constants.

2 Single-Source Shortest Paths

A common graph problem is to find the shortest path between two vertices in a graph G where weights have been assigned to the edges. Currently there is no known algorithm for this specific problem, but it is possible to efficiently find the shortest paths from some vertex v to every other vertex in the graph. From this, it is obviously very simple to solve our original problem. The new problem is called the single-source shortest path problem and this lecture presents algorithms for solving it.[2]

Define the *weight* of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The *shortest-path weight* can then be defined as

$$\delta(u, v) = \begin{cases} \min\{w(p) | u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Before proceeding, we examine the issue of negative-weight edges. The existence of a negative-weight edge does not affect the existence of a shortest-path weight. However, if there is a *cycle* $c = \langle v_0, v_1, \ldots, v_k, v_0 \rangle$ with a negative total weight, any path to any vertex reachable from c is undefined, as any proposed shortest path could be made even shorter by adding a traversal of the cycle c. We define $\delta(s, v) = -\infty$ if there is a negative-weight cycle on some path from s to v.

[2] This problem is discussed in chapter 24 of CLRS.

2.1 Relaxation

The algorithms presented here are based on the technique of *relaxation*. For each vertex v we maintain the *shortest-path estimate* $d[v]$, an upper bound on the weight of a shortest path from the source s to v. We relax on the edge (u, v) by updating $d[v]$ with $\min\{d[v], d[u] + w(u, v)\}$ and by appropriately updating the path itself. We initialize all shortest-path estimates to ∞ except $d[s]$ itself, where s is the source vertex. Since no edges need to be traversed to get from s to s, we set $d[s]$ to 0.

2.2 Dijkstra's Algorithm

Dijkstra's algorithm efficiently solves the single-source shortest path problem, but it requires the assumption that the graph contains no negative weight edges. The algorithm itself is simple; using a priority queue (we assume the use of a Fibonacci heap for efficiency), it repeatedly selects the vertex v with the smallest shortest-path estimate and relaxes on all edges incident from v. The DECREASE-KEY function is used to adjust the shortest-path estimate for a vertex if it needs to be changed. Since the algorithm executes at most $|E|$ DECREASE-KEY operations and $|V|$ EXTRACT-MIN operations, it requires a total of $O(E+V \log V)$ time. If each vertex stores the edge which caused it to be relaxed most recently, the actual shortest paths can be determined as well as their costs.

How do we know that Dijkstra's algorithm yields correct shortest paths? Say it does not yield correct results—the actual length of the shortest path to some vertex is not the distance computed by Dijkstra's algorithm. Consider the first vertex v for which $d[v] \neq \delta(s, v)$. When v was removed from the priority queue, it was the vertex with the lowest shortest-path estimate. Perhaps, however, there is some path to v that is shorter than $d[v]$. This path must go from the set of vertices which have already been extracted from the priority queue to the set of vertices still in the priority queue. Let x be the first vertex along this path still in the priority queue at the time v was removed from the queue. Since the algorithm greedily chose v rather than x, we know that $d[x] \geq d[v]$. Thus, the path through x to v has a length at least as large as $d[v]$ since we have assumed that the graph has nonnegative edge weights. Thus, the selection of v must not have been an error after all and the algorithm correctly identifies the minimum path lengths.

2.3 The Bellman-Ford Algorithm

The correctness of Dijkstra's algorithm depends crucially on the nonexistence of negative-weight edges. If there are negative-weight edges, we have to use another algorithm. One possibility is the Bellman-Ford algorithm. Bellman-Ford starts by initializing the estimated distance to the start vertex to 0 and the estimated distance to all other vertices in the graph to infinity. It then performs relaxations in $|V| - 1$ steps, relaxing every edge of the graph during each step.

First we show that this algorithm correctly identifies the minimum path lengths. If the graph contains no negative-weight cycles, we prove by induction that, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from s to v, where $s = v_0$ and $v = v_k$, then for $i = 0, 1, \ldots, k$, $d[v_i] = \delta(s, v_i)$ after the ith iteration and this property is maintained after the ith iteration. The base case, that $d[s] = \delta(s, s) = 0$ initially and at all points thereafter, is immediately clear. Inductively, assume that $d[v_{i-1}] = \delta(s, v_{i-1})$ after the $(i-1)$st iteration. In the ith iteration, the edge from v_{i-1} to v_i is relaxed, so we know that $d[i] = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) = \delta(s, v_i)$ afterwards. Since the longest possible path from s to any vertex in the graph visits $|V| - 1$ other vertices, the Bellman-Ford algorithm yields correct results provided the graph does not contain negative-weight cycles.

What happens if there is a negative-weight cycle? If there is a negative-weight cycle, there is no shortest path to at least some vertices of the graph, so additional relaxations beyond $|V| - 1$ will continue to cause shortest-path estimates to change. To detect negative-weight cycles, then, the algorithm performs one extra relaxation iteration. If any distance estimates change, the algorithm reports the existence of a negative-weight cycle.

Although this algorithm is more general than Dijkstra's algorithm, it is less efficient, running in $O(VE)$ time. In essence, to allow for negative-weight edges, we dispense with the greedy strategy used in Dijkstra's algorithm and hence must perform more work.

2.4 Special Case: Directed Acyclic Graphs

In addition to the general algorithms given above, we would like to present an algorithm which works very efficiently on a special case of the single-source shortest path problem. If the graph is a directed acyclic graph (also called a DAG), then topologically sorting[3] the vertices and relaxing their outgoing edges in this order will correctly identify the shortest paths. The rationale for this is the same as in the proof of correctness for Bellman-Ford; relaxing each edge along a path in the correct order identifies the shortest paths. By topologically sorting the vertices, we insure that the edges along each path are relaxed in the proper order. The running time is also very low; vertices in a graph can be topologically sorted in $O(V + E)$ time while the actual relaxation takes only $O(E)$ time. Thus, we can solve this special case of the single-source shortest path problem in only $O(V + E)$ time.

We have discussed the problem of finding the shortest paths from a particular source s to all other vertices in the graph. We came up with Dijkstra's algorithm which computes these shortest paths through greedy selections and relaxations. We saw that by using Fibonacci heaps, we could implement Dijkstra's algorithm in time:

$$O(V \log V + E)$$

We now pose the question of finding the shortest paths between all pairs of vertices in a graph. As a natural solution, we may simply run Dijkstra's algorithm individually with every vertex as a source. This will require running the algorithm $|V|$ times, giving a running time of:

$$O(V^2 \log V + EV)$$

We may similarly run Bellman-Ford's algorithm $|V|$ times to give the inferior running time of:

$$O(V^2 E)$$

2.5 Representation

Can we do better than this? Well, first we have to understand the problem and figure out how to represent its answer. Suppose we have n vertices. Then the input to our algorithm can be thought to be an $n \times n$ adjacency matrix whose (i, j)'th entry $w_{i,j}$ is:

$$w_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \text{ is not an edge of the graph} \\ \text{the weight of edge } (i,j) & \text{if } (i,j) \text{ is an edge of the graph} \end{cases}$$

Our output comes in two parts. First we need to know the distance between the pairs of vertices. We can represent this by a distance matrix D whose (i, j)'th entry $d_{i,j}$ is the length of the shortest path from i to j. However, this representation does not tell us the actual vertices on these shortest paths. For that

[3]It is assumed that the reader is familar with topological sorting. If this is not the case, you should review it in section 23.4 of the text

information, we need to return a predecessor matrix Π whose (i, j)'th entry $\pi_{i,j}$ is `NIL` if there is no path from i to j, and otherwise points to a predecessor of j on the shortest path from i to j.

As an example, suppose you had a graph where the shortest path from vertex 1 to 5 went through vertices 6, 2, and 8, in that order, and had weight 4.0. Then $d_{1,5} = 4.0$ and $\pi_{1,5} = 8; \pi_{1,8} = 2; \pi_{1,2} = 6; \pi_{1,6} = 1$.

2.6 Computation

We can approach the problem of computing the distance matrix by using dynamic programming. Specifically, we can define intermediate matrices D^m whose elements $d^m_{i,j}$ represent the length of the shortest path from i to j using at most m edges. Consider a path from vertex i to vertex j that uses at most m edges. If we take out the last edge in this path, call it edge (k, j), then we are left with a path of at most $m - 1$ edges from i to k. This is the crux of our dynamic programming: we consider all possible final edges in the path, and minimize over them (taking into account the possibility that the path has at most $m - 1$ edges as well):

$$d^m_{i,j} = \min\left\{d^{m-1}_{i,j}, \min_{1\le k\le n}\{d^{m-1}_{i,k} + w_{k,j}\}\right\}$$

The time needed to compute this expression is $\Theta(n)$, dictated by the number of terms we need to compare. However, for the dynamic programming to work, we need to compute n matrices $D^1, D^2, \ldots, D^n$, each of which contains n^2 elements, giving an overall running time of $O(n^4)$ which is already worse than what we had by repeating Dijkstra's algorithm.

A better dynamic programming strategy notices that any path from i to j that uses at most $2m$ edges will be comprised of a path for at most m edges from i to k and another path of at most m edges from k to j for some vertex k. Thus, we may formulate our recursive computation differently:

$$d^{2^m}_{i,j} = \min_{1\le k\le n}\{d^{2^{m-1}}_{i,k} + d^{2^{m-1}}_{k,j}\}$$

This way, we only need to compute $\lg n$ matrices D^{2^m}, giving a running time of $O(n^3 \log n)$, which is still worse than our repeated Dijkstra implementation.

The Floyd-Warshall takes an even craftier dynamic programming approach. Instead of making matrices according to the number of edges used in a path, it examines paths according to which vertices are used in them. Specifically, we redefine our distance matrix to be Δ^k whose (i, j)'th entry $\delta^k_{i,j}$ contains the length of the shortest path from i to j that goes through only vertices $\{1, 2, 3 \ldots k\}$. Now, the shortest path from i to j either uses vertex k or not. In the former case, our path from i to j can be divided into two paths, one from i to k and another from k to j, each of which only use vertices $\{1, 2, 3 \ldots k - 1\}$. Thus, the dynamic programming expression becomes:

$$\delta^k_{i,j} = \min\{\delta^{k-1}_{i,j}, \delta^{k-1}_{i,k} + \delta^{k-1}_{k,j}\} \tag{1}$$

The beauty of this is that the minimum is taken over only two elements, thus requiring $\Theta(1)$ time per term. We still have to compute n matrices Δ^k, and each has n^2 elements, giving an overall running time of $O(n^3)$. We can compute the matrices Π of predecessors with a similar recursive relationship where we compute predecessors in the shortest paths rather than lengths. Completing the details of such an approach serves as a good exercise and may be found in CLRS as Exercise 25.2-3 (page 699).

2.7 Negative Weight Edges

So far we have implicitly assumed that our graph has no negative-weight edges. In the case of negative-weight edges, Johnson's algorithm works by re-weighing the edges. Specifically, consider adding a huge positive number to each edge; this would ensure that there are no negative-weight edges, but it might change the shortest path because it introduces a dependence on the number of edges in the path. As another good exercise, design a graph whose shortest paths change when the same positive weight is added to each edge.

The idea behind Johnson's algorithm is to reweigh all edges in such a way that shortest paths use exactly the same vertices after the re-weighing as they did before. Specifically, we start by adding a new source s and edges of weight 0 from s to every node in the old graph. We now run Bellman-Ford's algorithm from the source s and record minimum distances $d(i)$ to any vertex i. If Bellman-Ford reports a negative-weight cycle, there is no solution to the problem as we may infinitely cycle around reducing shortest paths.

On the other hand, if Bellman-Ford does not report a negative-weight cycle, then we can take every edge (u, v) of weight w and make it have weight $w - d(v) + d(u)$. Clearly, the new weights must all be positive, because if $w - d(v) + d(u) \leq 0$ then $d(u) + w \leq d(v)$ contradicting our assertion that $d(v)$ is the shortest path from s to v. Moreover, in any path, the weights telescope. Thus, for example, a path progressing from vertex 1 to 3 to 8 to 2 will have total weight:

$$\begin{aligned} w_{1,3} - d(3) + d(1) &+ w_{3,8} - d(8) + d(3) + w_{8,2} - d(2) + d(8) \\ &= w_{1,3} + w_{3,8} + w_{8,2} + d(1) - d(2) \end{aligned}$$

Thus, consider two paths from i to j, path p_1 of weight w_1 and path p_2 of weight w_2. After the re-weighing, these paths will have weights $w_1 - d(j) + d(i)$ and $w_2 - d(j) + d(i)$ respectively. Thus, if p_1 had less weight before the reweighing, it will do so also after the reweighing, thus preserving all shortest paths. Overall then, we need to run Bellman-Ford once, adjust all the edge weights, and then run Dijkstra's algorithm on the new graph of non-negative-weight edges. This gives an overall running time of $O(V^2 \log V + VE)$.

As a final note, we will see in the upcoming lectures that the problem of finding the longest path between two vertices is a much more difficult problem, one we generally think cannot be done in polynomial time. Why is it that we cannot just take the negatives of all edges and do a shortest-path search? This is for the same reason that we had to do a more careful reweighing for Johnson's algorithm.