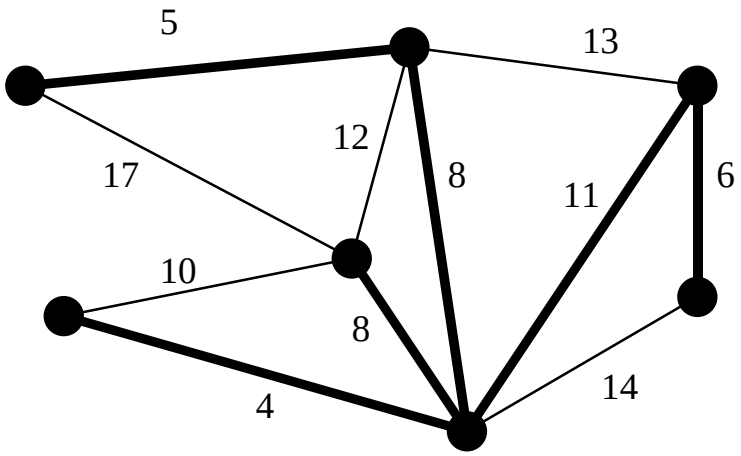
CS174 Lecture 17

Minimum Spanning Trees

Remember the minimum spanning tree problem from CS170 – you are given a graph G with weighted edges (real values on each edge) and the goal is to find a spanning tree T whose total weight is minimal. The minimum spanning tree is the least expensive way to connect up all the nodes. In the graph below, the minimum spanning tree is shown with heavy shaded edges.



Minimum spanning trees are useful for designing networks: computer networks, communication networks, and distribution networks for utilities like electrical power, gas, water etc. The edges represent links between sites, and the edge weight is the cost of connecting those sites. The minimum spanning tree is the cheapest structure that connects all the sites. You usually want some redundancy (extra paths) in such a network but the min spanning tree is a good starting point for designing an inexpensive network.

Deterministic Algorithms

You have hopefully seen two deterministic algorithms for computing the minimum spanning tree: Prim's and Kruskal's algorithms. Both are greedy algorithms. They build the spanning tree by adding at each step the minimum weight edge that satisfies a certain property. Specifically:

Prim's Algorithm

Input: G = (V,E) which is a connected graph
Output: T, the minimum spanning tree
T ← empty graph
For i = 1 to |V| do
  Let e be the minimum weight edge in G that touches T, and does not form a cycle with T.
  T ← T + {e}

It is fairly easy to see how to implement this algorithm efficiently by maintaining a heap of the edges incident to T that haven't been added yet, ordered by weight. If the graph has m edges and n vertices, the running time is O(m log n). The other deterministic algorithm is Kruskal's:

Kruskal's Algorithm

Input: G = (V,E) which is a connected graph
Output: T, the minimum spanning tree
T ← empty graph
For i = 1 to |V| do
  Let e be the minimum weight edge in G that does not form a cycle with T.
  T ← T + {e}

The only difference between the two is that Kruskal doesn't require the edge e to be connected to the evolving tree T. That means that T isn't necessarily connected at intermediate steps in Kruskal's algorithm. So strictly speaking the T in Kruskal's algorithm is a forest and not a tree. Kruskal's algorithm can also be implemented easily in O(m log n) time.

From looking at Prim and Kruskal, you can see that the general principle is similar. You pick the minimum weight edge that doesn't create a cycle. The order that you pick these edges doesn't matter much: whether they are minimum over all edges or just the edges incident to T.

e.g. suppose instead that for some forest T, you pick any vertex v not in T and add the minimum weight edge incident on v that doesn't create a cycle. Then you are safe, because that edge must be in the minimum spanning tree.

The advantage of the last method is that you can do it in parallel. Suppose that T is empty for now. For every vertex v in the graph, find the minimum weight edge incident on v, and add it to T. All of these edges must be in the minimum spanning tree. You don't need to check for cycles here because this scheme for choosing edges can't possibly create a cycle. (Problem: prove that the set of minimum weight edges incident on all vertices cannot contain a cycle).

If T is not empty though, you do need to check for cycles, which complicates the algorithm and makes it harder to implement in parallel. Ideally, you would only deal with the empty T case. But there is a way to do that – by contracting the edges you have added. Contraction preserves the minimum spanning tree in a certain sense (you should be able to prove that yourself):

Lemma 1

Let T be the minimum spanning tree of G, and let F be a subset of edges of T. Then if G' is the graph that results by contracting the edges F, and T' is the minimum spanning tree of G', then T is the union of the edges in F and the edges in T' (actually the "preimages" of the edges in T' because their endpoints may have changed through contractions).

Contraction effectively makes the partial spanning forest T disappear, and so you can reapply the parallel edge selection once again. This technique is called Boruvka's algorithm. It could be implemented like this:

Boruvka's Algorithm

Input: G = (V,E) which is a connected graph
Output: T, the minimum spanning tree
For each v in V do
  Let e be the minimum weight edge that is incident on v
  $T \leftarrow T + \{e\}$
$G' \leftarrow$ G with all edges in T contracted
$T' \leftarrow$ recursively compute the minimum spanning tree of G'
Return T + T'

First of all, notice that there are at least n/2 edges in T (or they could not touch all n vertices). So G' has at least n/2 contracted edges, and therefore at most n/2 vertices. The recursion depth is therefore $\log_2 n$. We claim that we can perform all the contractions at one level in O(m) time, where m is the number of edges in G. You should be able to verify this by reviewing the data structures to support contraction that we described last lecture. So overall, this algorithm has a running time of O(m log n) again.

But it would be nice if we could achieve O(m) running time (or O(n+m) to compute the minimum spanning forest for an unconnected graph). The weakness of Boruvka is that it reduces the number of vertices, but not necessarily the number of edges in each phase. You might have noticed that the number of edges is $O(n^2)$, and since n is decreasing, so is the upper bound on number of edges. That is true, but if m is "small" to begin with, e.g. O(n), then it takes O(log n) recursive steps before the upper bound caused by n will necessarily restrict the number of edges (the number of vertices has to decrease to about $\sqrt{n}$). Thus the running time really is O(m log n).

If we could somehow reduce the number of edges by some constant factor before the recursive call, then the problem sizes in the recursive call sequence would be a decreasing geometric series and the running time would be a constant times m. We will use probabilistic techniques to reduce the number of edges, and interleave those phases with Boruvka phases to reduce the number of vertices. First we need a notion of light and heavy edges.

Light and Heavy Edges

Let F be any forest of edges in a graph G. For any two vertices u and v, let $w_F(u,v)$ denote the maximum weight of any edge along the unique path in F from u to v. If there is no path, $w_F(u,v) = \infty$. Note that the normal weight of the edge in G between u and v is w(u,v). Then

If $w(u,v) > w_F(u,v)$, we say the edge {u,v} is **F-heavy**.

If $w(u,v) \leq w_F(u,v)$, then we say the edge {u,v} is **F-light**.

In particular, all the edges in F are F-light. Also, every edge in G is either F-heavy or F-light.

In order to reduce the number of edges in the graph, we will use the fact that a random subgraph of G has a "similar" minimum spanning tree. To be more precise, let G(p) be the subgraph of G obtained by adding each edge of G independently at random with probability p. we can say that

Lemma 2 Let F be the minimum spanning forest in the random graph G(p) obtained by keeping edges of G with probability p. Then the expected number of F-light edges in G is at most n/p.

The proof of this lemma is rather long and we wont give it here.

We can delete all the F-heavy edges in G without changing the spanning tree of G. That's because any edge that is F-heavy is also T-heavy ($w_T(u,v)$ must be less than or equal to $w_F(u,v)$, or T would not be a minimum spanning tree). Deleting a T-heavy edge doesn't change the spanning tree of G, which contains only T-light edges.

So we can reduce the spanning tree calculation for G to a spanning tree calculation for a graph with a *linear* number of edges (n/p). That is the essence of the linear-time algorithm. Here it is:

Algorithm MST

Input: weight graph G with n vertices and m edges

Output: Minimum spanning forest F for G.

1. Use three applications of Boruvka phases with contractions of edges. That produces a graph G_1 with at most n/8 vertices. Let C be the edges contracted during these phases. If G_1 is a single vertex, then exit and return F=C.
2. Let $G_2 = G_1(p)$ be a randomly sampled subgraph of G_1 with p = ½.
3. Recursively applying algorithm MST, compute the minimum spanning forest F_2 of the graph G_2.
4. Using a linear-time algorithm, find the F_2-heavy edges in G_1 and delete them to obtain a graph G_3.
5. Recursively apply algorithm MST to compute the minimum spanning forest F_3 for G_3.
6. Return the forest $F = C \cup F_3$.

Lemma 3 Algorithm MST runs in expected time O(n+m)

Proof:

Let T(n,m) denote the running time of the algorithm for a graph with n vertices and m edges.

At step 1, the three invocations of Boruvka run in deterministic time O(n+m).

Sampling the graph G_1 at step 2 takes O(m) time.

The graph G_2 has n/8 vertices and an expected number of edges which is at most m/2. So the cost of the recursive call at step 3 is T(n/8,m/2).

Step 4 claims a linear-time algorithm for removing F-heavy edges. This is tricky because you have to compare the edge weight for each {u,v} to the length of a shortest path in F from u to v.

We wont get into the details of how to accomplish that here, but we will use the fact that it can be done in time O(n+m).

Step 5 is applied to the graph G_3, which by lemma 2 has n/8 vertices and n/8p which is n/4 edges. The cost of this step is therefore T(n/8, n/4).

Adding up all these steps gives us the recurrence:

$T(n,m) \leq T(n/8, m/2) + T(n/8, n/4) + c(n+m)$.

A solution satisfying this recurrence is 2c(n+m), which proves the result that the expected running time of MST is O(n+m).