4 Minimum Spanning Trees—January 28-February 3, 2005

Introduction

For all of these notes, $G = (V, E)$ is an weighted, undirected, simple graph with n (unisolated) vertices and m edges. Assuming no isolated vertices gives $m \geq n/2$. To understand the Minimum Spanning Tree algorithm, we begin with some combinatorics.

Definition A *forest* is a graph with no cycles.

Definition A *tree* is a connected forest.

Definition A *spanning tree for* G is a tree with the same vertex set as G which forms a subgraph of G.

Definition A *spanning forest for* G is a forest with the same vertex set as G which forms a subgraph of G.

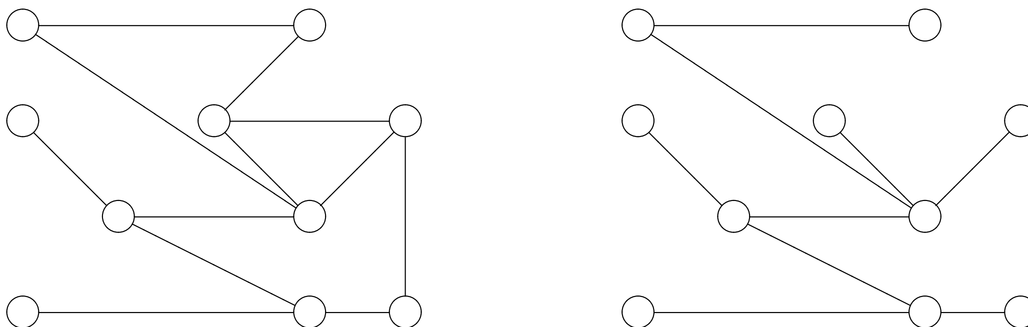

Figure 1: A graph G and a spanning tree for G.

For any graph G and any edge e which is not in the graph, we denote by $G + e$ the graph which has the same vertex set as G and all the edges of G as well as the edge e. This is an abuse of notation, but the meaning is relatively clear.

Theorem 4.1 *For any vertices $u \neq v$ in a tree T, there is a unique cycle between u and v.*

Theorem 4.2 *If T is a spanning tree of G and e an edge in G which is not in T, then the graph $T + e$ has an unique cycle.*

Proof Suppose we add the edge $e = \{u, v\}$ to the tree T. There is a path P between u and v in T, so P together with e forms a cycle. If there are two cycles in $T + e$, the edge e must be in both of them since T is a tree. Together, these cycles form two distinct paths between u and v in T, contradicting the previous theorem.

We denote the unique cycle in the graph $T + e$ by $\mathrm{cyc}_T(e)$.

Theorem 4.3 *(Switch Theorem) Let T be a spanning tree of a graph G and e an edge of G which is not in T. For any edge $f \in \mathrm{cyc}_T(e)$, the graph $T + e - f$ is a spanning tree.*

Proof Removing the edge f from $T+e$ destroys the unique cycle in this graph, so $T+e-f$ is a forest. It remains to show that it is connected. Let u, v be two vertices in T. Since T is a tree, there is a path between u and v. This path either includes the edge f, or it does not. If it does not include f, then it remains in $T+e-f$. If it does include f, we have a walk between u and v by replacing f in the previous path by $\mathrm{cyc}_T(e)-f$. Therefore, this new graph is connected, which shows that it is a tree (since we already have that it is a forest).

This is enough background to state the problem for which we are trying to find an algorithmic solution. For input, we have our weighted graph G as in the introduction. We denote the weight of the edge e by w_e. For any subgraph H of G we denote by $w(H)$ the weight of H and say that $w(H)=\sum_{e\in H} w_e$. We define T_m, the *Minimum Spanning Tree* of G to be the spanning tree of least weight for G. As an exercise, we will show that if the edges have pairwise distinct weights (no two edges have the same weight) then there is a unique spanning tree of minimum weight, so we refer feel safe calling it the minimum spanning tree rather than a minimum spanning tree. We abbreviate "minimum spanning tree" as MST.

Theorem 4.4 *For any graph G, the MST satisfies the following:*

- **The Cut Property or Blue Rule:** *For any cut $(X, \overline{X})$ of G, the lightest edge crossing the cut is in T_m.*
- **The Cycle Property or Red Rule:** *For all cycles C in G, the heaviest edge in C is not in T_m.*

Proof Red Rule: This is a proof by contradiction. Suppose the MST T_m of G contains the heaviest edge e in the cycle C. Let U be the graph $T_m - e$. Since T was a tree, removing e results in a graph with two connected components, U_1 and U_2. Removing e disconnected the graph so these vertices must be in different connected components. Likewise, since C is a cycle and connected, there must be another edge $f \neq e$ in the cycle with its endpoints in different connected components. Since the endpoints are in different connected components of U, it is clear that f is not in U. Also, since adding f connects two components of the forest U, $U+f$ is a tree. What is the weight of this tree?

$$\begin{aligned} w(U+f) &= w(T_m - e + f) \\ &= w(T_m) - w(e) + w(f) \\ &< w(T_m) \end{aligned}$$

This is a contradiction since T_m is the MST, so the heaviest edge in any cycle cannot be in the MST.

Blue Rule: Let $(X, \overline{X})$ is a cut of the graph G, and e the lightest edge crossing this cut. Suppose that e is not in the MST, T_m. There must be some f crossing this cut which is also in $\mathrm{cyc}_{T_m}(e)$. The graph $T' = T - e + f$ is also a spanning tree and $w(T') < w(T_m)$, contradicting that T_m is the MST.

Historical Algorithms

Historically, there have been many algorithms for finding the MST. The first, from 1926, is Borůvka's algorithm, giving a time bound of $O(\max\{n^2, m\log n\})$. This is also the first algorithm ever published. In 1930, Jarník gave an $O(m + n\log n)$ algorithm, which was later rediscovered, independently, by both Prim (1956) and Dijkstra (1958). The other historical algorithm was discovered by Kruskal in 1956, with time $O(m\log n)$.

More recent work by Yao gives an $O(m \log \log n)$ algorithm. Later, Fredman and Tarjan discribe an algorithm which runs in $O(m \log^* n)$, where $\log^*$ is the inverse of the tower function. Unsatisfied, Gabow and Gadil give an algorithm with $O(m \log \log^* n)$, and Chazelle worked out an algorithm with time $O(m\alpha(m, n))$ where $\alpha(m, n)$ is the inverse of the Ackerman function. Still, work on this problem continued until 1999 when Karger, Klein and Tarjan published an algorithm with $O(m)$ expected time. This randomized algorithm degenerates to Borůvka's algorithm in the worst case, and so has the $O(\max\{n^2, m \log n\})$ worst case running time.

The first three algorithms, which we will call Borůvka's, Prim's and Kruskal's algorithms, are the simplest to understand and will be covered here, as will the Karger *et al.* algorithm.

Kruskal's Algorithm

Start with the entire graph G.
Process edges in increasing order by weight.
For each edge:
  Discard it if retention would create a cycle among edges retained so far.

This algorithm is correct by the red rule.

Prim's Algorithm

Start with an empty forest, a dstinguished vertex a.
while (forest unconnected)
  Let T be the tree in the forest touching a.
  Grow T by the lightest edge across $V((T), \overline{V(T)})$

This algorithm is correct by the blue rule.

Borůvka's Algorithm

Start with an empty forest F.
while(unconnected)
  For each tree T in F
    Identify ligthest edge crossing $(V(T), \overline{V(T)})$
    Add this edge to F.
This algorithm is correct by the blue rule.

We call each iteration of the while loop in Borůvka's algorithm a *Borůvka phase*. Using graph theorhetic techniques, each phase take $O(m)$ time. Since each phase reduces the number of connected components by half, the total running time is $O(m \log n)$. The $O(n^2)$ running time is more complicated and thus not included here.

An alternative way of looking at a Borůvka phase is to contract the lightest edges. In this contraction, we retain only the lightest edge if multiple edges are created, thus retaining a simple graph.

A Linear Time Algorithm

The idea of the linear time algorithm that we are looking at is that Borůvka's algorithm only tells you what edges to *include* in the MST, and never gets rid of edges that cannot be in the MST. The best algorithm would take advantage of both the blue rule and the red rule. But how do we know when an edge can't be in the MST?

Definition For any forest F which is a subgraph of G and any vertices u, v in the graph, we define $w_F(u, v)$ to be the maximum weight on an edge on the unique path between u and v in F. If no

such path exists, we say that $w_F(u,v) = \infty$. Now, for any edge $e = \{u,v\}$ we call e *F-heavy* if $w_e > w_F(u,v)$. Otherwise, we call the edge *F-light*.

Note that edges between components of F can only be F-light since nothing is bigger than ∞. By the red rule, an edge that is F-heavy for any forest F is not in the MST, T_m. The best forest for throwing away edges would then be T_m, but we don't have T_m! So we try to get something close to T_m without taking a lot of time. The algorithm ends up looking like this.

Minimum Spanning Forest Algorithm for G

Step 1 Apply two Borůvka phases and record the contracted edges (E_1). Call the new graph G_1.

Step 2 Form a graph H by including each edge of G_1 independently with probability $1/2$. Recursively find the MSF of H and call it F.

Step 2.1 Identify the F-heavy edges of G_1 and delete them to form graph G_2.

Step 3 Recursively find the MSF of G_2 and call it F'. The MSF of G is $F \cup E_1$.

Since *Step 1* includes edges which are in the MSF by the blue rule and *Step 2.1* eliminates edges which are not in the MSF by the red rule, the algorithm finds the correct MSF. There are two recursive calls, so we have a binary tree of subproblems. At the first level, the left subproblem is to find the MSF of H and the right subproblem is to find the MSF of G_2. Since we have a binary tree, there are at most 2^d subproblems at a depth d in the tree. 2^{d-1} of these are left subproblems and 2^{d-1} are right subproblems.

For the running time, other than the recursive calls, all of these operations are clearly linear time in the number of edges except the identification of F-heavy edges in *Step 2.1*. This operation is also deterministically linear time using a modified MST verification algorithm, but it is beyond the scope of these notes and will appear in the next ones.

It only remains to analyze the total number of edges in the problem and all subproblems to get the running time of the algorithm.

Expected Running Time

To do the expected running time, we employ a technique called *left-child decomposition*. The tree is partitioned into subtrees where each subtree has as its root either the root of the main tree, or a right subproblem. The subtree consists of all of the nodes reachable by a series of left steps down the tree. We end up with a partition as in Figure 2.

If there are k edges in the root node of one of these subtrees, how many edges are expected in the whole subtree? If there are k edges in a parent graph, G_1, we can expect $k/2$ edges in the subgraph H generated in *Step 2*. Since there are less edges than this in G_1 (G_1 is generated from G by contraction, G has k edges), in the node at depth d from the root of the subtree we expect at most $k/2^d$ edges, for a total of at most $2k$ expected edges in the subtree. But what number k of edges can we expect in the root of the subtree?

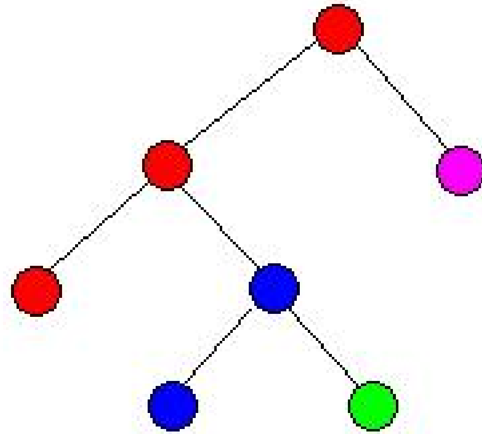

Figure 2: A binary tree that decomposes into 4 subtrees, one red, one green, one blue, and one violet.

Lemma 4.5 *For a subgraph H of G obtained by including each edge independantly with probability p, let F be the minimum spanning forest of H. The expected number of F-light edges in G is at most n/p.*

Proof For the proof, we process the edges in increasing order by weight, as in Kruskal's Algorithm. In this way, we form the subgraph H and the MSF F of H at the same time. For each edge, we first decide if it is F-heavy or F-light. This is done by testing whether the endpoints are in the same connected component of our current forest F. Once this is known, we flip a coin with probability p of coming up heads. If it lands heads, we include the edge in H. If, in addition, it is F-light, we include the edge in F.

Since only heavier edges are processed later, the F-heavy edges remain F-heavy and the F-light edges remain F-light. Since they are inconsequential to our analysis, we ignore all of the coin flips for F-heavy edges and concentrate on the F-light edges. There can be at most n edges in F, and F-light edges are in F on heads, so we see only F-heavy edges after seeing n heads. Thus the total number of F-light edges is the number of coin flips we have to do with a weight p coin before seeing n heads, which is exactly a negative binomial variable and has expected value n/p.

Using this Lemma, we expect to see $2v/4$ edges in a subproblem which has v vertices before *Step 1*. Since in a right subproblem at depth d, there are $n/4^d$ expected vertices, we have a total bound on the number of vertices in right subproblems as

$$\sum_{d=1}^{\infty} 2^{d-1}(2n)/4^d = n/2$$

Combining this with the expectation of $2k$ edges in all the left children of a right subproblem and m edges in the root, we get a bound of $O(m+n)$ on the expected number of edges in the entire problem.

The preceeding has been the proof of the following theorem:

Theorem 4.6 *The expected running time of this MSF algorithm is $O(m)$.*

Probability of Expected Case

Theorem 4.7 *The MSF algorithm runs in $O(m)$ time with probability* $1 - \exp(\Omega(m))$.

Proof The algorithm will fail to run in linear time if the number of edges in all the right subproblems is nonlinear in m or if the number of edges in all the left subproblems is nonlinear in the total number of edges in all right subproblems. So what is the probability of either one of these happening?

Edges in a right subproblem result from F-light edges in *Step 2*. The total number of F-light edges is the total number of coin flips as in the analysis for Lemma 4.5. The number of coin flips is bounded by the number of heads, which can be at most the number of vertices in all the right subproblems. There are at most $n/2$ vertices in all right subproblems, so the probability that less than $n/2$ heads occur in a sequence of $3m$ coin flips is the same as the probability that there are more than $3m$ edges in all the right subproblems. Since $m \geq n/2$, we know that this probability is $\exp(\Omega(m))$ by a Chernoff bound.

Let the number of vertices in all the right subproblems be m'. We now consider all the coin flips in *Step 2*, both for the F-heavy and F-light edges. The sequence of coin flips from the root of our subtree for a given edge are all heads and a terminating tails. Concatenating these coin flips for all edges, we see that the number of edges in all left subproblems is the number of heads in these sequences, while the number of edges in all right subproblems is the number of tails, so the probability that there are more than $3m'$ edges in all left subproblems is at most the probability that fewer than m' tails occur in a sequence of $3m'$ coin flips, which is $\exp(\Omega(m))$ by a Chernoff bound.

Combined, this gives a probability of linear time at least $1 - \exp(\Omega(m))$.

Worst-Case Analysis

While the probability of linear time is good, it is always possible that it will fail, so we want to estimate the worst-case running time. For this, we need to look at the total number of edges in two ways.

Consider a subproblem with k edges and v vertices. How many edges are in both subproblems combined? There are three kinds of edges that we can see in subproblems. In the left subproblem, we have the edges in H but not in F and edges in F. The edges that are not in F do not occur in the right subproblem, so the only edges that can occur in both subproblems are the edges in F. There are at most $v/4$ of these because they form a forest in the contrracted graph G_1. At least $v/2$ edges are removed in *Step 1*, so there are at most k edges in both subproblems.

Since we begin with m edges, the above analysis shows there are at most m edges at each level, for a total of $O(m \log n)$ edges.

The number of edges can also be bounded by the number of vertices, which is reduced by a factor of 4 at each level. This gives at most $n^2/4^d$ edges at each level, giving an $O(n^2)$ bound on the number of edges. Thus the worst case running time is $O(\min\{n^2, m \log n\})$.