COS 423 Minimum Spanning Trees
Spring 2007

1. The Generic Greedy Algorithm

The *generic greedy algorithm* finds a minimum spanning tree (MST) by an edge-coloring process. Initially all edges are uncolored. The algorithm colors edges blue (accepted) or red (rejected) one-at-a-time by repeatedly applying either of the following two coloring rules, in any order:

Blue Rule: Given a cut that no blue edges cross, select a minimum uncolored edge crossing the cut, and color it blue. (A *cut* is a partition of the vertices into two non-empty parts; an edge *crosses* the cut if it has one end in each part.)

Red Rule: Given a cycle containing no red edges, select a maximum uncolored edge on the cycle, and color it red.

Termination: As long as at least one edge is uncolored, some edge can be colored by one of the rules. That is, the algorithm eventually colors all the edges, no matter in what order it applies the rules.

Proof: Suppose some edge $\{v, w\}$ is uncolored. Let X be the set of vertices reachable from v by traversing only blue edges, and let $\overline{X}$ be the remaining vertices. If w is in X, then $\{v, w\}$ is on a cycle all of whose other edges are blue, and the red rule applies to color $\{v, w\}$ red. Otherwise, X, $\overline{X}$ is a cut that {v, w} but no blue edges cross. Thus the blue rule applies to color $\{v, w\}$ or some other edge blue. QED

Correctness: The algorithm maintains the invariant that some MST T contains all of the blue edges and none of the red ones.

Proof: By induction on the number of edges colored. The invariant is vacuously true initially, since no edges are colored and any MST satisfies the invariant. Suppose the invariant holds for an MST T before an application of the blue rule that colors a minimum edge $\{v, w\}$ crossing a cut X, $\overline{X}$. If $\{v, w\}$ is in T, the invariant holds for T after $\{v, w\}$ is colored. Otherwise, consider the cycle formed by adding $\{v, w\}$ to T. This cycle contains at least one edge $\{x, y\}$ in T crossing X, $\overline{X}$. (It may contain several.) Edge $\{x, y\}$ must be uncolored since it crosses the cut, so it has cost at least as large as $\{v, w\}$. But adding $\{v, w\}$ to T and deleting $\{x, y\}$ forms a new spanning tree T', which has cost no more than that of T. Since T is minimum, T' must also be minimum (and $\{v, w\}$ and $\{x, y\}$ must have the same cost). Tree T' satisfies the invariant after $\{v, w\}$ is colored.

On the other hand, suppose the invariant holds for an MST T before an application of the red rule that colors a maximum edge $\{v, w\}$ on a cycle C. If $\{v, w\}$ is not in T, the

invariant holds for T after $\{v, w\}$ is colored. Otherwise, deleting $\{v, w\}$ from T produces two trees. At least one edge $\{x, y\}$ of C other than $\{v, w\}$ must have exactly one end in each of these trees. Edge $\{x, y\}$ cannot be blue because it is not in T, and it cannot be red since C contains no red edges, so it must be uncolored. Since the red rule applies to color $\{v, w\}$, $\{v, w\}$ has cost no less than that of $\{x, y\}$. But deleting $\{v, w\}$ from T and adding $\{x, y\}$ forms a new tree T'. Since T is an MST, T' must also be an MST and $\{v, w\}$ and $\{x, y\}$ must have equal cost. Tree T' satisfies the invariant after $\{v, w\}$ is colored. QED

The invariant implies that once all edges are colored (or even once the blue edges form a single tree), the blue edges form an MST. The proof also implies that if all edge costs are distinct, there is a unique MST. (Why?)

2. Three Classical Algorithms

The generalized greedy algorithm is highly non-deterministic in that the coloring rules can be applied in any order. There are three classical MST algorithms, two of them well-known and one much-less so, that are specializations of the generic algorithm. In describing these algorithms, I shall use the term "blue tree" to refer to any of the maximal trees defined by the blue edges. (By the invariant, the blue edges can never contain a cycle.) Initially there are n one-vertex blue trees. Each time an edge is colored blue, two blue trees are combined into one, until after $n-1$ edges are colored blue, there is only one blue tree, which is an MST.

2.1. Kruskal's Algorithm

Kruskal's algorithm, presented in 1956, is globally greedy: examine the edges in non-decreasing order by cost. When examining an edge $\{v, w\}$, color it blue if its ends are in different blue trees and red otherwise. (Prove that this algorithm is a specialization of the generic algorithm.)

To implement Kruskal's algorithm efficiently, we need two things: a mechanism for selecting edges in non-decreasing order by cost, and a way to keep track of the vertex sets of the blue trees. The latter is an instance of the disjoint set union problem, and the compressed-tree data structure solves it. There are at most two finds per edge and exactly $n-1$ links, for a total time of $O(m\alpha(m, n))$. To select edges in non-decreasing order by cost, we can either sort the edges by cost or use a heap. If we use a binary-comparison-based method, the time bound for selecting edges will be $O(m \log m) = O(m \log n)$. Using sorting has the advantage that the constant factor is small; using a heap has the advantage on a dense graph that we may have to delete only a small fraction of the edges from the heap before the MST is completed, resulting in a running time smaller than the worst-case bound. One intriguing possibility is to use an incremental version of quicksort that only does work as needed to determine the next edge to process. Such a method has a low constant factor and takes advantage of early completion of the MST. The bottleneck in Kruskal's algorithm is the time to order the edges, unless the edges are given in sorted

order or can be sorted fast, for example by using radix sorting. Then the bottleneck is the set maintenance, and the overall running time is $O(m\alpha(m,n))$. rather than $O(m \log n)$.

2.2. The Algorithm of Jarnik, Prim, and Dijkstra

The other well-known classical algorithm is generally credited to Prim (1957) and Dijkstra (1959) independently, but was actually discovered by Jarnik(1930). The JPD algorithm is a single-source version of the generic algorithm: it grows a single blue tree from a fixed starting vertex s (the *seed*). The general step, repeated $n-1$ times, is to find a minimum-cost edge with exactly one end in the blue tree containing s and color it blue. (Prove that this algorithm is a specialization of the generic algorithm.) This algorithm can be implemented just like Dijkstra's single-source shortest-path algorithm; indeed, he presented both algorithms in the same paper. We maintain a set of labeled vertices, which are those adjacent to some vertex in the blue tree containing s. The key of a labeled vertex is the cost of a minimum edge connecting it to a vertex in the blue tree containing s. We initialize the set of labeled vertices to be those adjacent to s, with appropriate keys. The general step is to choose a labeled vertex v of minimum key, color blue a minimum edge connecting it to the blue tree containing s, and examine each edge $\{v,w\}$: if w is not labeled and not in the blue tree containing s, mark it labeled and set its key equal to the cost of $\{v,w\}$; if w is labeled and has key greater than the cost of $\{v,w\}$, set its key equal to the cost of $\{v,w\}$. If we use a Fibonacci heap to store the set of labeled vertices, the total running time of Kruskal's algorithm is $O(n \log n + m)$, because there are $n-1$ insert and $n-1$ delete min operations and at most m decrease-key operations.

2.3 Boruvka's Algorithm

Boruvka's algorithm, presented in 1926, is a concurrent version of the generic algorithm. It proceeds in passes. A pass consists of selecting, for each blue tree, a minimum edge with exactly one end in the blue tree, and concurrently coloring all the selected edges blue. An edge can be selected twice, once for the blue tree containing each end. This "algorithm" is actually not correct if edge costs can be equal (give a counterexample), but if the edge costs are distinct, or are made distinct by using a tie-breaking rule (such as numbering all the edges and breaking ties by edge number), then Boruvka's algorithm can be serialized into a specialization of the generic algorithm. (Prove this.)

One nice feature of Boruvka's algorithm is that it is simple to build an efficient implementation. A pass of Boruvka's algorithm can be performed in $O(m)$ time as follows: using graph search, find the vertex sets of the blue trees. For each edge, determine the blue trees of its endpoints. Make a pass through the edges, keeping track for each blue tree of a minimum edge with exactly one end in the tree. We can actually implement the algorithm using a disjoint set data structure, without graph search, so that each pass takes $O(m)$ time via a single pass through the edges (doing finds), followed by a pass over the edges to be colored blue (doing links). (Prove this.)

One can easily prove by induction that, after pass k of Boruvka's algorithm, each blue tree contains at least 2^k vertices (prove this), which implies that Boruvka's algorithm stops after at most $\lg n$ passes and runs in $O(m \log n)$ time at $O(m)$ time per pass.

By doing some additional work during each pass, specifically by applying the red rule to reject edges, one can improve the running time of Boruvka's algorithm on certain kinds of graphs. During a pass, we reject all edges both of whose ends are in the same blue tree, and also all but one edge, an edge of minimum cost, joining each pair of blue trees. Rejecting the edges with both ends in the same blue tree is an easy extension of the method described above for doing a pass. To reject all but a minimum-cost edge between each pair of blue trees, we label each edge with the blue trees containing its ends, sort the edges lexicographically by their labels (via a two-pass radix sort, which takes $O(m)$ time because the labels can be integers from 1 through n), and scan the edges in label-sorted order, keeping only a minimum edge in each group with equal labels. This sweep amounts to contracting each blue tree to a single vertex and deleting loops and multiple edges. We call this algorithm *Boruvka's algorithm with cleanup*. On general graphs, Boruvka's algorithm with cleanup runs in $O(n^2)$ time, because after the i^{th} pass only $(n/2^i)^2$ edges can remain. On families of sparse graphs closed under contraction, such as planar graphs, Boruvka's algorithm with cleanup runs in $O(m) = O(n)$ time. (Prove this for planar graphs.)

3. Faster Algorithms

The three classical algorithms described in the last section leave open the question of whether there is a sorting bottleneck in the minimum spanning tree problem; that is, can one beat $O(m \log n)$ on sparse (as opposed to dense) graphs, or might MST computation take $\Omega(n \log n)$ time, if only binary comparisons between edge weights are allowed? The answer is no: by combining two ideas from the previous section, we can do better. In particular, suppose we run $\lg \lg n$ passes of Boruvka's algorithm, which takes $O(m \log \log n)$ time, and then do a cleanup. This reduces the number of blue trees to at most $n / \lg n$. If we now choose one of these blue tree as a seed and run the JPD algorithm, treating each blue tree as a single vertex, then there will only be $n / \lg n$ delete-min operations, and if we use a Fibonacci heap the running time of this second part of the algorithm will be $O(m)$, resulting in an overall time bound of $O(m \log \log n))$.

3.1 Yao's Algorithm

An $O(m \log \log n)$ bound was actually obtained by Andy Yao before the invention of Fibonacci heaps, by adding to Boruvka's algorithm a simpler idea than Fibonacci heaps that I shall call *packets.* In order to select minimum edges incident to blue trees, we maintain a two-level data structure containing for each blue tree its incident uncolored edges. These edges are partitioned into *packets*, each of which contains at most $\lg n$ edges. A *full packet* starts out containing exactly $\lg n$ edges. In each packet we sort the

edges by cost. Initially we arbitrarily partition the edges incident to each vertex into packets, at most one non-full packet per vertex, and sort the packets. Each edge is initially in exactly two packets, one for each of its ends. To perform a Boruvka step, for each blue tree, we examine each of its packets in turn. For a particular packet, we remove edges in order until finding one that has an end in another blue tree, or until the packet is empty. Once we have found a minimum for each packet, or emptied it, we compare the packet minima, and select a minimum over all the packets. Having done this for all the blue trees, we color all the selected edges blue, deleting them from their packets, and combine the sets of packets of blue trees that are connected together. If the set of packets for a blue tree is represented as a circular list, then combining the sets of packets of newly connected blue trees takes $O(1)$ time per new blue edge.

To keep track of the vertex sets of the blue trees, we use a disjoint set data structure as in Kruskal's algorithm; the time needed for set operations is asymptotically less than that required for sorting. Ignoring the time for the set operations, the time for one pass of Boruvka's algorithm is $O(1)$ per packet. If all the packets were full, this time would be $O(m/\log n)$, summing to $O(m)$ time overall. The time to sort the packets, which is $O(m \log\log n)$ would dominate the running time.

This idea almost works, the catch being that there can be one non-full packet per vertex initially, and if these packets are not at some point combined into full packets, then the overall time to process non-full packets might be $O(n \log n)$. (But even so we get an overall bound of $O(n \log n + m \log\log n)$, better than Kruskal's algorithm and better than Boruvka's algorithm with cleanup.) We can handle this by breaking the computation into two parts, as in the Fibonacci-heap-based method described above: we run Boruvka's algorithm with the initial packets for $\log\log n$ passes. Then we reinitialize the packets, resulting in at most $n/\log n$ non-full packets, and run the algorithm to completion. The result is an algorithm running in $O(m \log\log n)$ time. (Prove this.)

3.2. Faster Algorithms Using Fibonacci Heaps, Packets, and Other Ideas

Thus we have two different ways to obtain an $O(m \log\log n)$ time bound. One might well suspect that if we can obtain an $O(m \log\log n)$ bound without using Fibonacci heaps, then one might be able to do even better using Fibonacci heaps. That is, the hybrid method described in the introduction might not be best possible. This is true: one can implement a generalized form of Boruvka's algorithm using Fibonacci heaps that has a time bound of $O(m \log^* n)$. Furthermore the idea of packets is independent of that of Fibonacci heaps and can be combined with it to yield a further improvement, to $O(m \log\log^* n)$ time. Even this has been improved: Chazelle, in a tour-de-force, using a new heap structure that is allowed to make errors (but only in a controlled way) obtained a deterministic $O(m\alpha(m,n))-$time algorithm. Furthermore, by using additional ideas, this algorithm can be modified to run in minimum time to within a constant factor, though no one can presently say what the asymptotic bound is. (It might be $O(m)$, it might be

$O(m\alpha(m,n))$, it might be something in between. How is this possible, you ask? The secret is to build (by brute-force search) an optimum algorithm for very small problem sizes, and use it in a rapidly-unwinding divide-and-conquer recurrence (which gets down to very small sub-problems in a constant number of recursive calls). These algorithms are all "beyond the scope of this course," as they say. But I'll offer one idea to think about, which may help you in solving the MST problem on Problem Set 3: consider a version of the simple two-pass algorithm described at the beginning of this section in which we run Boruvka passes, using a Fibonacci heap for each blue tree to store incident uncolored edges. Coloring an edge blue requires melding two such heaps. We run Boruvka passes until some appropriate stopping criterion holds, then do a cleanup, and then finish the computation using the JPD algorithm as at the beginning of the section. What can you say about the running time of such an algorithm, especially as applied to graphs of large girth?

3.3. A Randomized Linear-Time Algorithm

The holy grail for the MST problem is to find a linear-time algorithm. This has been done, with a caveat: the algorithm uses random sampling. The algorithm is quite simple, if one sweeps under the rug the details of one crucial part of the algorithm.

How might one find an MST in linear time? We know from the discussion in Section 2.3 that one pass of Boruvka's algorithm with cleanup will reduce the effective number of vertices by at least a factor of two. If we had a corresponding way, for a sufficiently dense graph, to reduce the number of edges by a constant factor in linear time, we would be able to find an MST in linear time by appropriately intermixing these steps: if the graph is sparse ($m < cn$ for some appropriate constant c) do a Boruvka step; otherwise, do an edge-thinning step. (Why would this result in a linear-time algorithm?)

The missing part is thus an edge-thinning method. For this we use a generalization of MST verification. Suppose we are given a tree T and asked to determine whether it is an MST. The red rule implies that T is an MST if and only if, for every non-tree edge $\{v,w\}$, $\{v,w\}$ is maximum on the cycle formed by adding $\{v,w\}$ to T. More generally, if T is *any* tree, or even any forest, we can apply the red rule to reject any edge {v, w} not in T if it forms a cycle with edges in T and it is maximum on this cycle. This gives us a way to do thinning, but for this method to reject a lot of edges (we want to reject a constant fraction), we must start with a good tree (or forest), perhaps not an MST but something good enough.

To find a good forest, we combine random sampling with recursion. Specifically, the edge-thinning step is as follows. Select a random sample of the edges by flipping a fair coin (one whose probably of coming up heads is one half) once for each edge and putting the edge in the sample if the coin comes up heads. (The expected size of the sample will be half the total number of edges.) Next, find a minimum spanning forest F of the sample by applying the whole algorithm recursively(!). (The sample may not define a connected subgraph, so we may not get a spanning tree of the whole graph, only a spanning tree of each connected component of the sample.) Reject (by the red rule) all the sampled edges

not in F. Test every edge not in the sample against F, and reject it if it forms a cycle with edges in F and it is maximum on the cycle.

Now we have all the pieces (except for an efficient way to test edges not in the sample against F, which I shall discuss later). The overall algorithm, which we present as an algorithm to find a minimum spanning forest (one tree per connected component) of a not-necessarily connected graph, consists of repeating the appropriate one of the following cases until no edges remain, and then returning the set of edges in the original graph corresponding to those colored blue by Boruvka steps (since each Boruvka step contracts the graph, the edges colored blue in later Boruvka steps are not in the original graph but in the current contracted and thinned graph; nevertheless, they correspond to original edges): if the graph contains fewer than cn edges, do a Boruvka step; otherwise, sample the edges, compute an MSF F for the sample by applying the entire algorithm recursively, and thin the graph using F and the red rule to reject every edge not in F that forms a cycle with F and is maximum on the cycle.

Thinning Efficiency: The expected number of edges *not* rejected by a thinning step is at most $2n$. That is, the expected number of edges left after a thinning step is at most $2n$. (Thus for example if $c = 4$ then thinning will on the average discard at least half the edges.)

Proof: We use Kruskal's algorithm, but for purposes of the analysis only. Consider the following process for doing the sampling, building F, and rejecting edges: sort the edges in non-decreasing order by cost. Process the edges in order. To process an edge $\{v, w\}$, flip the coin, resulting in H or T. If $\{v, w\}$ connects two vertices in the same blue tree, reject it (color it red). Otherwise, if the coin is H, color the edge blue: it is in the sample and it belongs in F. Otherwise, leave it in the graph: it is not in the sample and it cannot be rejected. This process gives exactly the same forest F and exactly the same set of rejected edges as the algorithm. (Why?) But notice that if an edge is rejected, the outcome of the coin flip is irrelevant! Thus we can modify the procedure, without affecting its behavior, so that it first tests an edge to see if it forms a cycle and only flips the coin if it does not. Now the edges that are not rejected are exactly those for which the coin is flipped. Each time the coin is flipped, an H results in the addition of an edge to F. But F can contain at most $n-1$ edges. Thus the sequence of non-thinned edges corresponds exactly to a sequence of coin flips containing at most $n-1$ heads. What is the expected length of such a sequence? It is at most $2n$. (Prove this; it follows from very basic results in discrete probability.) QED

One can prove by setting up a suitable recurrence that if the time to do the sampling and thinning is $O(m)$, then the expected running time of the overall algorithm is $O(n+m)$, if c is chosen appropriately. (Do this. What is a good value of c?)

The only missing piece of the puzzle is how to do the thinning. This problem is solvable by an extension of path compression: union by rank or size cannot be used and something more complicated is needed. This approach results in a time bound for thinning of $O(m\alpha(m,n))$. This bound can be improved to $O(m)$ by adding the idea of building an

optimum algorithm for very small problem sizes. I omit the details of these ideas (at least here).

This result leaves open a number of interesting questions. A minor one is exactly how to put the several pieces of the algorithm together in the best way to give the best constant factor or the simplest algorithm. The most complicated part of the algorithm is the piece I have not described, the thinning test. It would be nice to have a simpler solution for this; perhaps randomization would help. Finally, is there a deterministic linear-time algorithm?