CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Th. Feb. 5, 1998

Notes by Samir Khuller.

1 Minimum Spanning Trees

Read Chapter 6 from [?]. Yao's algorithm is from [?].

Given a graph $G = (V, E)$ and a weight function $w : E \rightarrow R^+$ we wish to find a spanning tree $T \subseteq E$ such that its total weight $\sum_{e \in T} w(e)$ is minimized. We call the problem of determining the tree T the minimum-spanning tree problem and the tree itself an MST. We can assume that all edge weights are distinct (this just makes the proofs easier). To enforce the assumption, we can number all the edges and use the edge numbers to break ties between edges of the same weight.

We will present now two approaches for finding an MST. The first is Prim's method and the second is Yao's method (actually a refinement of Boruvka's algorithm). To understand why all these algorithms work, one should read Tarjan's *red-blue* rules. A red edge is essentially an edge that is not in an MST, a blue edge is an edge that is in an MST. A *cut* (X, Y) for $X, Y \subset V$ is simply the set of edges between vertices in the sets X and Y.

Red rule: consider any cycle that has no red edges on it. Take the highest weight uncolored edge and color it red.

Blue rule: consider any cut $(S, V - S)$ where $S \subset V$ that has no blue edges. Pick the lowest weight edge and color it blue.

All the MST algorithms can be viewed as algorithms that apply the red-blue rules in some order.

1.1 Prim's algorithm

Prim's algorithm operates much like Dijkstra's algorithm. The tree starts from an arbitrary vertex v and grows until the tree spans all vertices in V. At each step our currently connected set is S. Initially $S = \{v\}$. A lightest edge connecting a vertex in S with a vertex in $V - S$ is added to the tree. Correctness of this algorithm follows from the observation that a partition of vertices into S and $V - S$ defines a cut, and the algorithm always chooses the lightest edge crossing the cut. The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by edges in MST. Using a Fibonacci heap we can perform EXTRACT-MIN and DELETE-MIN operation in $O(\log n)$ amortized time and DECREASE-KEY in $O(1)$ amortized time. Thus, the running time is $O(m + n \log n)$.

It turns out that for sparse graphs we can do even better! We will study Yao's algorithm (based on Boruvka's method). There is another method by Fredman and Tarjan that uses F-heaps. However, this is not the best algorithm. Using an idea known as "packeting" this The FT algorithm was improved by Gabow-Galil-Spencer-Tarjan (also uses F-heaps). More recently (in Fall 1993), Klein and Tarjan announced a linear time randomized MST algorithm based on a linear time verification method (i.e., given a tree T and a graph G can we verify that T is an MST of G?).

1.2 Yao/Boruvka's algorithm

We first outline Boruvka's method. Boruvka's algorithm starts with a collection of singleton vertex sets. We put all the singleton sets in a queue. We pick the first vertex v, from the head of the queue and this vertex selects a lowest weight edge incident to it and marks this edge as an edge to be added to the MST. We continue doing this until all the vertices in the queue, are processed. At the end of this round we contract the marked edges (essentially merge all the connected components of the marked edges into single nodes – you should think about how this is done). Notice that no cycles are created by the marked edges if we assume that all edge weights are distinct.

Each connected component has size *at least* two (perhaps more). (If v merges with u, and then w merges with v, we get a component of size three.) The entire processing of a queue is called a phase. So at the end of phase i, we know that each component has at least 2^i vertices. This lets us bound the number of phases by $\log n$. (Can be proved by induction.) This gives a running time of $O(m \log n)$ since there are at most $\log n$ phases. Each phase can be implemented in $O(m)$ time if each node marks the cheapest edge incident to it, and then we contract all these edges and reconstruct an adjacency list representation for the new "shrunken" graph where a connected component of marked edges is viewed as a vertex. Edges in this graph are edges that connect vertices in two different components. Edges between nodes in the same component become self-loops and are discarded.

We now describe Yao's algorithm. This algorithm will maintain connected components and will not explicitly contract edges. We can use the UNION-FIND structure for keeping track of the connected components.

The main bottleneck in Boruvka's algorithm is that in a subsequent phase we have to recompute (from scratch) the lowest weight edge incident to a vertex. This forces us to spend time proportional to $\sum_{v \in T_i} d(v)$ for each tree T_i. Yao's idea was to "somehow order" the adjacency list of a vertex to save this computational overhead. This is achieved by partitioning the adjacency list of each vertex v into k groups $E_v^1, E_v^2, \ldots, E_v^k$ with the property that if $e \in E_v^i$ and $e' \in E_v^j$ and $i < j$, then $w(e) \leq w(e')$. For a vertex with degree $d(v)$, this takes $O(d(v) \log k)$ time. (We run the median finding algorithm, and use the median to partition the set into two. Recurse on each portion to break the sets, and obtain four sets by a second application of the median algorithm, each of size $\frac{d(v)}{4}$. We continue this process until we obtain k sets.) To perform this for all the adjacency lists takes $O(m \log k)$ time.

Let T be a set of edges in the MST. VS is a collection of vertex sets that form connected components. We may assume that VS is implemented as a doubly linked list, where each item is a set of vertices. Moreover, the root of each set contains a pointer to the position of the set in the doubly linked list. (This enables the following operation: given a vertex u do a FIND operation to determine the set it belongs to, and then yank the set out of the queue.)

The root of a set also contains a list of all items in the set (it is easy to modify the UNION operation to maintain this invariant.) $E(v)$ is the set of edges incident on vertex v. $\ell[v]$ denotes the current group of edges being scanned for vertex v. small$[v]$ computes the weight of the lowest weight edge incident on v that leaves the set W. If after scanning group $E_v^{\ell[v]}$ we do not find an edge leaving W, we scan the next group.

```
proc Yao-MST(G);
    T, VS ← ∅;
    ∀v : ℓ[v] ← 1;
    while |VS| > 1 do
        Let W be the first set in the queue VS;
        For each v ∈ W do;
            small[v] ← ∞;
            while small[v] = ∞ and ℓ[v] ≤ k do
                For each edge e = (v, v') in E_v^{ℓ[v]} do
                    If find(v') = find(v) then delete e from E_v^{ℓ[v]}
                     else small[v] ← min (small[v], w(e))
                If small[v] = ∞ then ℓ[v] ← ℓ[v] + 1
            end-while
        end-for
        Let the lowest weight edge out of W be (w, w') where w ∈ W and w' ∈ W';
        Delete W and W' from VS and add union(W, W') to VS;
        Add (w, w') to T
    end-while
end proc;
```

Each time we scan an edge to a vertex in the same set W, we delete the edge and charge the edge the cost for the find operation. The first group that we find containing edges going out of W, are the edges we pay for (at most $\frac{d(v)}{k} \log^* n$). In the next phase we start scanning at the point that we stopped last time. The overall cost of scanning in one phase is $O(\frac{m}{k} \log^* n)$. But we have $\log n$ phases in all, so the total running time amounts to $O(\frac{m}{k} \log^* n \log n) + O(m \log k)$ (for the preprocessing). If we pick $k = \log n$ we get $O(m \log^* n) + O(m \log \log n)$, which is $O(m \log \log n)$.