

Notes by Samir Khuller.

7 Minimum Spanning Trees

Read Chapter 6 from [21]. Yao's algorithm is from [23].

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.

7.1 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 ?).

7.2 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 “shrunk” 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 $E_v^1, E_v^2, \dots, E_v^k$ 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 . $\text{small}[v]$ 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 $O(m \log \log n)$.

Notes by Samir Khuller.

8 Fredman-Tarjan MST Algorithm

Fredman and Tarjan's algorithm appeared in [6].

We maintain a forest defined by the edges that have so far been selected to be in the MST. Initially, the forest contains each of the n vertices of G , as a one-vertex tree. We then repeat the following step until there is only one tree.

High Level

```

start with  $n$  trees each of one vertex
repeat
    procedure GROWTREES
    procedure CLEANUP
until only one tree is left
    
```

Informally, the algorithm is given at the start of each round a forest of trees. The procedure GROWTREES grows each tree for a few steps (this will be made more precise later) and terminates with a forest having fewer trees. The procedure CLEANUP essentially “shrinks” the trees to single vertices. This is done by simply *discarding* all edges that have both the endpoints in the same tree. From the set of edges between two different trees we simply retain the lowest weight edge and discard all other edges. A linear time implementation of CLEANUP will be done by you in the homework (very similar to one phase of Boruvka's algorithm).

The idea is to grow a single tree only until its heap of neighboring vertices exceeds a certain critical size. We then start from a new vertex and grow another tree, stopping only when the heap gets too large, or if we encounter a previously stopped tree. We continue this way until every tree has grown, and stopped because it had too many neighbours, or it collided with a stopped tree. We distinguish these two cases. In the former case refer to the tree has having stopped, and in the latter case we refer to it as having halted. We now condense each tree into a single supervertex and begin a new iteration of the same kind in the condensed graph. After a sufficient number of passes, only one vertex will remain.

We fix a parameter k , at the start of every phase – each tree is grown until it has more than k “neighbours” in the heap. In a single call to Growtrees we start with a collection of *old trees*. Growtrees connects these trees to form *new* larger trees that become the *old trees* for the next phase. To begin, we *unmark* all the trees, create an empty heap, pick an unmarked tree and grow it by Prim's algorithm, until either its heap contains more than k vertices or it gets connected to a marked old tree. To finish the growth step we mark the tree. The F-heap maintains the set of all trees that are adjacent to the current tree (tree to grow).

Running Time of GROWTREES

Cost of one phase: Pick a node, and mark it for growth until the F-heap has k neighbors or it collides with another tree. Assume there exist t trees at the start and m is the number of edges at the start of an iteration.

$$\text{Let } k = 2^{2m/t}.$$

Notice that k increases as the number of trees decreases. (Observe that k is essentially 2^d , where d is the average degree of a vertex in the super-graph.) Time for one call to Growtrees is upperbounded by

$$\begin{aligned}
 O(t \log k + m) &= O(t \log(2^{2m/t}) + m) \\
 &= O(t 2m/t + m) \\
 &= O(m)
 \end{aligned}$$

This is the case because we perform at most t deletemin operations (each one reduces the number of trees), and $\log k$ is the upperbound on the cost of a single heap operation. The time for CLEANUP is $O(m)$.

Data Structures

Q = F-heap of trees (heap of neighbors of the current tree)
 e = array[trees] of edge ($e[T]$ = cheapest edge joining T to current tree)
mark = array[trees] of (true, false), (true for trees already grown in this step)
tree = array[vertex] of trees (tree[v] = tree containing vertex v)
edge-list = array[trees] of list of edges (edges incident on T)
root = array[trees] of trees (first tree that grew, for an active tree)

```
proc GROWTREES(G);
  MST  $\leftarrow$   $\emptyset$ ;
   $\forall T$  : mark[ $T$ ]  $\leftarrow$  false;
  while there exists a tree  $T_0$  with mark[ $T_0$ ] = false do
    mark[ $T_0$ ]  $\leftarrow$  true; (* grow  $T_0$  by Prim's algorithm *)
     $Q \leftarrow \emptyset$ ; root[ $T_0$ ]  $\leftarrow T_0$ 
    For edges  $(v, w)$  on edge-list[ $T_0$ ] do;
       $T \leftarrow$  tree[ $w$ ]; (* assume  $v \in T_0$  *)
      insert  $T$  into  $Q$  with key =  $w(v, w)$ ;
       $e[T] \leftarrow (v, w)$ ;
    end-for
    done  $\leftarrow (Q = \emptyset) \vee (|Q| > k)$ ;
    while not done do
       $T \leftarrow$  deletemin( $Q$ );
      add edge  $e[T]$  to MST; root[ $T$ ]  $\leftarrow T_0$ ;
      If not mark[ $T$ ] then for edges  $(v, w)$  on edge-list[ $T$ ] do
         $T' \leftarrow$  tree[ $w$ ]; (* assume  $v \in T$  *)
        If root[ $T'$ ]  $\neq T_0$  then (* edge goes out of my tree *)
          If  $T' \notin Q$  then insert  $T'$  into  $Q$  with key =  $w(v, w)$  and  $e[T'] \leftarrow (v, w)$ ;
          else if  $T' \in Q$  and  $w(v, w) < w(e[T'])$  then dec-key  $T'$  to  $w(v, w)$  and  $e[T'] \leftarrow (v, w)$ ;
        end-if
      done  $\leftarrow (Q = \emptyset) \vee (|Q| > k) \vee$  mark[ $T$ ];
      mark[ $T$ ]  $\leftarrow$  true;
    end-while
  end-while
end proc;
```

Comment: It is assumed that *insert* will check to see if the heap size exceeds k , and if it does, no items will be inserted into the heap.

We now try to obtain a bound on the number of iterations. The following simple argument is due to Randeep Bhatia. Consider the effect of a pass that begins with t trees and $m' \leq m$ edges (some edges may have been discarded). Each tree that stopped due to its heap's size having exceeded k , has at least k edges incident on it leaving the tree. Let us "charge" each such edge (hence we charge at least k edges).

If a tree halted after colliding with an initially grown tree, then the merged tree has the property that it has charged at least k edges (due to the initially stopped tree). If a tree T has stopped growing because it has too many neighbors, it may happen that due to a merge that occurs later in time, some of its neighbors are in the same tree (we will see in a second this does not cause a problem). In any case, it is true that *each* final tree has charged at least k edges. (A tree that halted after merging with a stopped tree does not need to charge any edges.) Each edge may get charged twice; hence $t' \leq \frac{2m'}{k}$. Clearly, $t' \leq \frac{2m}{k}$. Hence $k' = 2 \frac{2m}{t'} \geq 2^k$. In the first round, $k = 2^{2m/n}$. When $k \geq n$, the algorithm runs to completion. How many rounds does this take? Observe that k is increasing exponentially in each iteration.

Let $\beta(m, n) = \min\{i \mid \log^{(i)} n \leq \frac{2m}{n}\}$. It turns out that the number of iterations is at most $\beta(m, n)$. This gives us an algorithm with total running time $O(m\beta(m, n))$. Observe that when $m > n \log n$ then $\beta(m, n) = 1$. For graphs that are not too sparse, our algorithm terminates in one iteration! For very sparse graphs ($m = O(n)$) we actually run for $\log^* n$ iterations. Since each iteration takes $O(m)$ time, we get an upper bound of $O(m \log^* n)$.

CLEANUP will have to do the work of updating the root and tree data structures for the next iteration.