15-859E: Advanced Algorithms **CMU, Spring 2015**
Lecture #2: Randomized MST and MST Verification January 14, 2015
Lecturer: Anupam Gupta Scribe: Yu Zhao

1 Preliminaries

In this lecture we are talking about two contents: Karger-Klain-Tarjan algorithm and MST Verification. Our setting is the same as last lecture. Let $G = (V, E)$ to be a simple graph with vertex set V and edge set E, where $|V| = n$ and $|E| = m$. We also assume that all the weights of edges are distinct. In Lecture 1 we saw some deterministic algorithms finding MST in time $O(m \log n)$, $O(m + n \log n)$ and $O(m \log^* n)$. Our goal in this lecture is to find MST in the graph in $O(m + n)$ expected time. Remember the two rules we mentioned in Lecture 1: cut rule and cycle rule.

1.1 Heavy & light

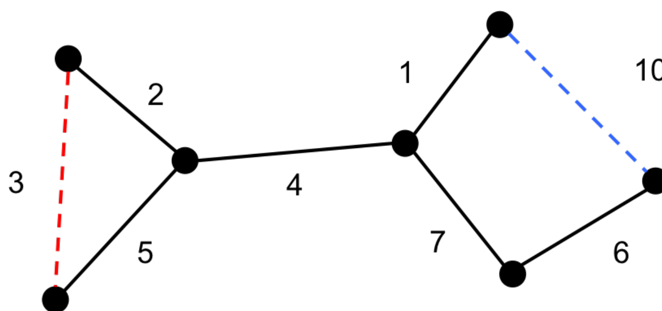

Figure 2.1: Example of heavy and light edges

Take any tree[1] in a graph. For example, in Figure 2.1 we pick the tree with black edges. The tree might not be an MST, or even not be connected (a forest). Now look at any other edge in the graph. The blue edge on the right side is heavier than any other edge in the cycle. Then by cycle rule this edge will not be in the MST of the graph. What about the red edge on the left side? This edge is not the heaviest edgei n the cycle, and in fact this edge appears in the MST of the graph. Therefore we can discard some edges which will never appear in the MST based on any given tree of a graph.

Definition 2.1. Let T be a forest of graph G and $e \in E(G)$. If e create a cycle when adding to T and e is heaviest edge in the cycle, then we call edge e is T-*heavy*. Otherwise, edge e is T-*light*.

Every edge should be either T-heavy or T-light. Notice that if an edge e does not make a cycle, then e is T-light. If edge e is in the tree T, edge e is also T-light. In Figure 2.1, the red edge is T-light, the blue edge is T-heavy, and all black edges are T-light.

Fact 2.2. *Edge e is T-light iff $e \in MST(T \cup \{e\})$.*

[1] Here when we talk about trees, in fact we mean trees or forests. And MST means the minimum spanning forest if the graph is not connected.

Fact 2.3. *If T is a MST of G then edge $e \in E(G)$ is T-light iff $e \in T$.*

Therefore our idea is that pick a good tree/forest T, find all the T-heavy edges and get rid of them. Hopefully the number of edges remained is small. Then we will find the MST in remaining edges. We want to find some tree T such that there are a lot of T-heavy edges.

1.2 Magic Blackbox

How should we find all the T-heavy edges? Here we first assume a magic blackbox: MST Verification. We'll show how to implement MST Verification later.

Theorem 2.4 (MST Verification)**.** *Given a tree $T \subseteq G$, we can output all the T-light edges in $E(G)$ in time $O(|V| + |E|)$.*

This wonderful blackbox can output all the T-light edges in linear time.

2 Karger-Klain-Tarjan Algorithm

Suppose a graph $G = (V, E)$ has n vertices and m edges. Here we will present the algorithm KKT(G) as following.

1. Run 3 rounds of Borůvka's Algorithm on G to get a graph $G' = (V', E')$ with $n' \le n/8$ vertices and $m' \le m$ edges.
2. $E_1 \leftarrow$ a random sample of edges E' of G' where each edge is picked independently with probability 1/2.
3. $T_1 \leftarrow \text{KKT}(G_1 = (V', E_1))$.
4. $E_2 \leftarrow$ all the T_1-light edges in E'.
5. $T_2 \leftarrow \text{KKT}(G_2 = (V', E_2))$.
6. Return T_2 (combine with the edges chosen in Step 1).

Here the idea is that we randomly choose half of the edges and find the MST on those edges. We hope this tree T will have a lot of T-heavy edges therefore we can discard these edges and find the MST on the remaining graph.

Theorem 2.5. *KKT algorithm will return MST(G).*

Proof. We can prove it by induction. For the basic case it is trivial, because we will find the MST in Step 1. Otherwise in Step 3 we find a MST T_1 of graph G_1, and in E_2 just discard all the T_1-heavy edges from the entire edges E' (not only from E_1!!) which are not possible to be in the MST of G'. Or in other words, $\text{MST}(G') \subseteq E_2$. Therefore what Step 5 returns is the MST of G_2, which is also the MST of G'. Combine it with the edges we choose in Step 1, we get the MST of graph G. □

The key idea of this proof is that discarding heavy edges of any tree in a graph will remain MST the same.

Now we need to deal with the complexity.

Claim 2.6. $\mathbf{E}[\#E_1] = \frac{1}{2}m'$.

This claim is trivial since we pick each edge with probability 1/2.

Claim 2.7. $\mathbf{E}[\#E_2] \leq 2n'$. *This means graph G_2 will be hopefully very sparse.*

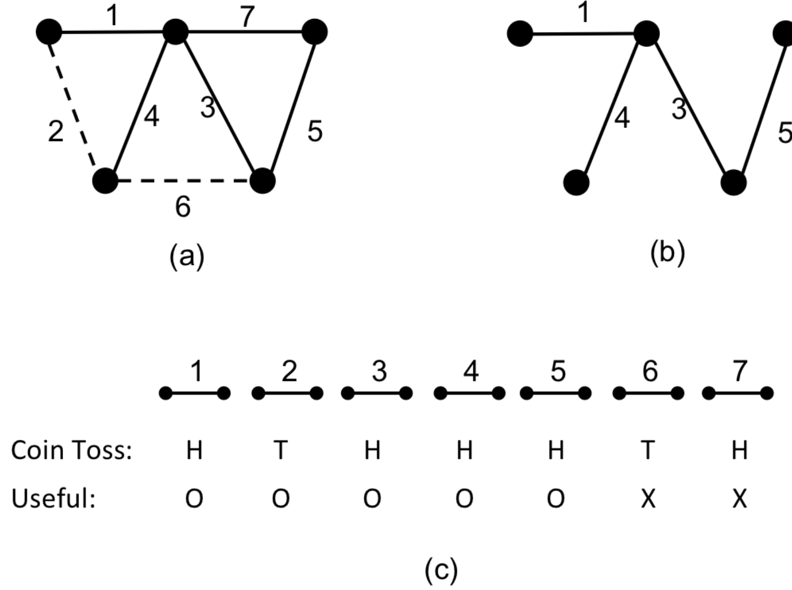

Figure 2.2: Illustration of another order of coin tossing

How should we analysis $\mathbf{E}[\#E_2]$? In Step 2 of KKG algorithm, we first flip a coin on each edge, then take all the edges tossing head and finally in Step 3 we use KKG algorithm on this new graph to get the MST of G_1.

Let's think of this process in another order. First of all, Step 3 is to calculate the MST of the new graph. It does not matter if we use KKG algorithm or some other method, and we will still get the same MST. Then instead of first flipping coins for all edges then calculating the MST. We can do these two things simultaneously: We flip the coins for the edges in an increasing order by weight, and do Kruskal immediately after any edge tossing head.

We define a coin toss is *useful*, if Kruskal algorithm would like to add this edge in the MST, and define a coin toss is *useless* if not. For example, Figure 2.2(a) gives a graph with coin toss result. The solid edges are the edges with tossing head, and the dashed edges are the edges with tossing tail. Figure 2.2(b) is the MST on all edges tossing head. Now let's check the usefulness of all these coin flips, as in Figure 2.2(c). In the beginning the MST is an empty set. Then the coin flip of the first edge is useful, since it should be added into the MST. The tossing result is head, so we add this edge into MST. The coin flip of the second edge is useful, since it should be added into the MST. However the tossing result is tail, so sadly we can not add this edge into MST. The coin flip of the third, forth, fifth edges are all useful, and the tossing result are all head, so we add all of them into the MST. The coin toss of sixth edge is useless, because we will not add this edge into MST since it will creat a cycle, so is the seventh edge.

Claim 2.8. *$T_1 = MST(G_1)$. $e \in E'$ is T_1-light if and only if the coin flip of e is useful.*

Proof. If the coin flip of e is useful, then at the moment we flip the coin by adding e into the MST there is no cycle, which means either $e \in T_1$ (coin tossing head), or there is no cycle in $T_1 \cup \{e\}$, or edge e is not the heaviest edge in the cycle of $T_1 \cup \{e\}$. In any of these cases, edge e is T_1-light. On the other hand, if the coin flip of e is useless, then at the moment we flip the coin there would be a cycle if we add e into the MST, therefore edge e is T_1-heavy.

□

Then we can prove Claim 2.7.

Proof of Claim 2.7.

$$\mathbf{E}[\#E_2] = \mathbf{E}[\#\text{useful coin flips}] \leq \frac{n'-1}{1/2} \leq 2n'$$

The first inequality holds since by each useful coin flip we may add an edge into the MST of G_1 with probability $1/2$, and the final $T_1 = MST(G_1)$ has at most $n'-1$ edges. Therefore the expectation of useful coin flips is at most $2(n'-1) \leq 2n'$. □

Theorem 2.9. *KKT$(G=(V,E))$ can return the MST in time $O(m+n)$.*

Proof. Let X_G be the expect running time on graph G, and

$$X_{m,n} := \max_{G=(V,E),|V|=n,|E|=m} \{X_G\}$$

In KKG algorithm, Step 1, 2, 4 and 6 will be done in linear time, so we assume the time cost of Step 1, 2, 4 and 6 is at most cm. Step 3 will spend time X_{G_1}, and Step 5 will spend time X_{G_2}. Then we have

$$X_G \leq cm + X_{G_1} + X_{G_2} \leq cm + X_{m_1,n'} + X_{m_2,n'}$$

Here we assume that $X_{m,n} \leq c(2m+n)$, then

$$\begin{aligned} X_G &= cm + \mathbf{E}[c(2m_1+n')] + \mathbf{E}[c(2m_2+n')] \\ &\leq c(m+m'+6n') \\ &\leq c(2m+n) \end{aligned}$$

The first inequality holds because $\mathbf{E}[m_1] \leq \frac{1}{2}m'$ and $\mathbf{E}[m_2] \leq 2n'$. The second inequality holds because $n' \leq n/8$ and $m' \leq m$. □

3 MST Verification

Now we come back the implement of the magic blackbox. Here we only consider about the trees (not forests). Here we refine Theorem 2.4 as following.

Theorem 2.10 (MST Verification). *Given $T=(V,E)$ where $|V|=n$ and m pairs of vertices (u_i,v_i), we can find the heaviest edge on the path in T from u_i to v_i for all i in $O(m+n)$ time.*

Tarjan find an algorithm in Time $O(m\alpha(m,n))$ using Union-Find. Komlos find how to do it with $O(m+n)$ comparisons. Dixon-Rauch-Tarjan on RAM machine with O(m). Here we will give the idea of Komlos.

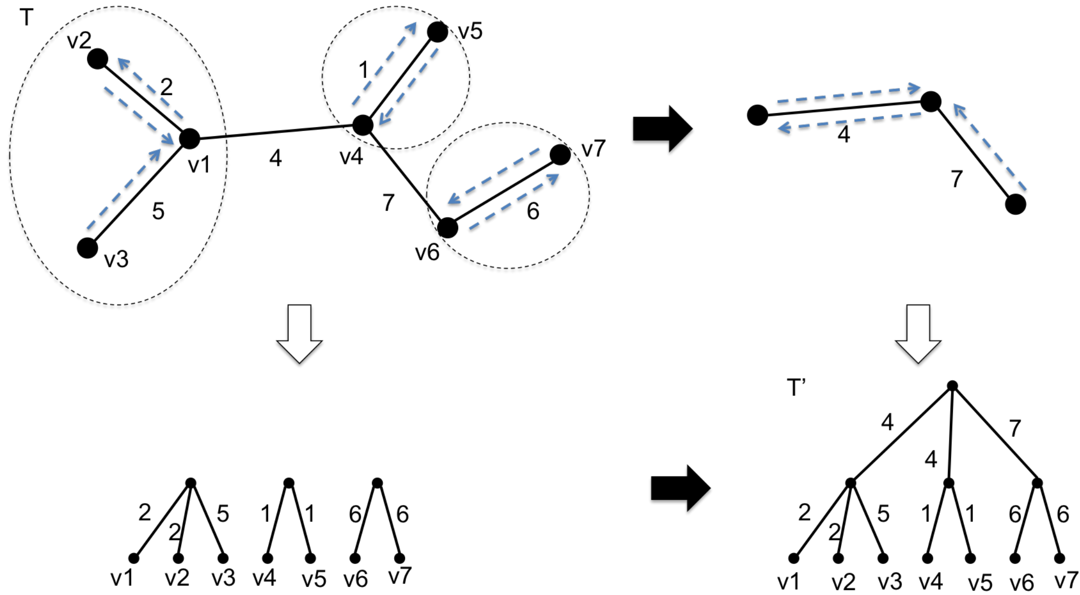

Figure 2.3: Illustration of balancing a tree

3.1 Balance the tree

Suppose $T = (V, E)$ is a tree on n vertices, and we run Boruvka's algorithm on T. (Let $V_1 = V$ be the original vertices at the beginning of round 1, V_i be the vertices in round i, and say there are L rounds so that $|V_{L+1}| = 1$). We build a tree T' as follows: the vertices are the union of all the V_i. There is an edge from $v \in V_i$ to $w \in V_{i+1}$ if the vertex v belongs to a component in round i and that is contracted to form $w \in V_{i+1}$; the weight of this edge (v, w) is the min-weight edge out of v in round i. (Note that all vertices in V are now leaves in T'.) Figure 2.3 shows an example of balancing a tree.

Fact 2.11. *For nodes u, v in a tree T, let $\texttt{maxwt}_T(u, v)$ be the maximum weight of an edge on the (unique) path between u, v in the tree T. For all $u, v \in V$, we have*

$$\texttt{maxwt}_T(u, v) = \texttt{maxwt}_{T'}(u, v)$$

This is a problem in Homework 1. In Figure 2.3, we have $\texttt{maxwt}_T(v_1, v_7)$ is 7 which is the weight of edge (v_4, v_6). We can check that $\texttt{maxwt}_{T'}(v_1, v_7)$ is also 7.

Fact 2.12. *T' has at most $\log_2 n$ layers since each vertex has at least 2 children and all leaves of T' are at the bottom (the same) level..*

Here we can make a balanced tree T' based on tree T with some good properties. Then we can just query the heaviest edges of vertex pairs in tree T' instead of T. Another trick is that we can assume that all queries are ancestor-descent queries if we can find the least common ancestor quickly.

Theorem 2.13 (Harel-Tarjan)**.** *Given a tree T, we can preprocess in $O(n)$ time, and answer all LCA queries in $O(1)$ time.*

3.2 Get the answer

Now we have reduced our question to how to answer the ancestor-descent queries efficiently. For each edge $e = (u, v)$ where v is the parent of u, we will look at all queries starting in subtree T_u and ending above vertex v. Say those queries go to $w_1, w_2, \ldots, w_k$. Then the "query string" is $Q_e = (w_1, w_2, \ldots, w_k)$. Then we need to calculate the "answer string" $A_e = (a_1, a_2, \cdots, a_k)$ where a_i is the maximal weight among the edges between w_i and u.

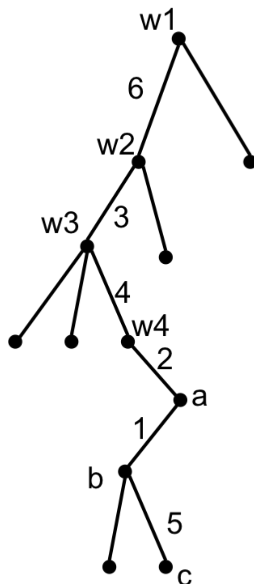

Figure 2.4: Illustration of queries and answers

Figure 2.4 gives us an example. Suppose $Q_{(b,a)} = (w_1, w_3, w_4)$, which means there are three queries starting from some vertices in the subtree of b and ending to w_1, w_3, w_4. Then we get the answer

$$A_{(b,a)} = (a_1, a_3, a_4) = (6, 4, 4)$$

since the maximal weight of an edge on the path from w_1 to b is the weight of edge (w_1, w_2), and the maximal weight of an edge on the path from either w_3, w_4 to b is from the weight of edge (w_3, w_4).

Given the answer of $A_{(b,a)}$, how should we find the answer of $A_{(c,b)}$ efficiently? Say $Q_{(c,b)} = (w_1, w_4, b)$. Comparing with $Q_{(b,a)}$, we may lose some queries since they are from some other children of vertex b, and we may have some other queries ending at b which will not contain in $Q_{(b,a)}$. The easiest way is to update every query. Suppose the weight of edge (c, b) is t. Then we can calculate

$$A_{(c,b)} = (\max\{a_1, t\}, \max\{a_4, t\}, t) = (\max\{6, 5\}, \max\{4, 5\}, 5)$$

The trick is that if in $Q_e = (w_1, w_2, \ldots, w_k)$, $w_1, w_2, \ldots, w_k$ is sorted from the top to the bottom. than the answers $A_e = (a_1, a_2, \cdots, a_k)$ should be non-increasing, $a_1 \geq a_2 \geq \cdots \geq a_k$. Therefore we can do binary search to reduce the number of comparisons.

Claim 2.14. *Given the question string Q_e for $e = (u, v)$ where v is the parent of u, the answer string A_e for e , we can compute answers $A_{e'}$ for $e' = (w, u)$ where w is a child of u, within time $\lceil \log(|A_e| + 1) \rceil$.*

Theorem 2.15. *The total number of comparisons for all queries*

$$t \le \sum_e \log\left(|Q_e| + 1\right) \le O(m+n)$$

Proof. Assume the number of edges in level i is n_i. Here the level is counted from the bottom to the top, the edges connected to leaves are at level 0.

$$\begin{aligned}
\sum_{e \in \text{level } i} \log_2(1+|Q_e|) &= n_i \operatorname*{avg}_{e\in\text{level } i}\left(\log_2(1+|Q_e|)\right) \\
&\le n_i \log_2\left(1 + \operatorname*{avg}_{e\in\text{level } i}(|Q_e|)\right) \\
&= n_i \log_2\left(1 + \frac{\sum_{e\in\text{level } i}|Q_e|}{n_i}\right) \\
&\le n_i \log_2\left(1+\frac{m}{n_i}\right) \\
&= n_i\left(\log_2\frac{m+n}{n} + \log_2\frac{n}{n_i}\right)
\end{aligned}$$

The first inequality holds by Jensen's inequality and convexness of function $\log_2(1+x)$. The second inequality holds since the number of all queries is m and each query will only appear on at most one edge on any particular level. Therefore we have

$$\begin{aligned}
t &= \sum_e \log_2(1+|Q_e|) \\
&= \sum_i \sum_{e\in\text{level } i} \log_2(1+|Q_e|) \\
&\le \sum_i n_i\left(\log_2\frac{m+n}{n} + \log_2\frac{n}{n_i}\right) \\
&= n\log_2\frac{m+n}{n} + \sum_i n_i \log_2\frac{n}{n_i} \\
&\le n\log_2\frac{m+n}{n} + cn \\
&= O(n\log\frac{m+n}{n} + n) = O(m+n)
\end{aligned}$$

□