## Lecture 8: Dynamic Programming on Trees and 2-Trees

In Lecture 7, we noted that every 2-tree contains a *2-leaf*, i.e. a vertex which is adjacent to only 2 other vertices, which are themselves adjacent . More strongly, we noted that every 2-tree which is not a triangle contains a 2-leaf $v$ such that $G-v$ is a 2-tree. This allowed us to prove by induction that every 2-tree had a planar straight line embedding.

In this lecture, we observed that this fact also allows us to prove by induction that $G$ is a 2-tree precisely if it has at least 3 vertices, and we can order its vertices as $v_1,\ldots,v_n$ so that for each $i>2$, the set $\{v_j|j<i\}$ contains exactly two vertices, which are joined by an edge. Using this result as a starting point , we then made the following:

***Observation:*** for every 2-tree $G$ and 2-leaf $v$, there is an ordering of V(G) as $v_1,\ldots,v_n$ so that (i) for each $i>2$, the set $\{v_j|j<i\}$ has precisely two vertices which are joined by an edge, and (ii) $v$ is $v_n$. Hence $G-v$ is a 2-tree.

***Proof:*** Choose an ordering of V(G) as $v_1,\ldots,v_n$ so that for each $i>2$, the set $\{v_j|j<i\}$ has precisely two vertices, which are joined by an edge, so as to maximize the value of $k$ for which $v=v_k$. Note that the condition on our ordering implies that $v_1,v_2,v_3$ is a triangle and hence $v_2$ is adjacent to $v_1$ and $v_3$. We want to show $v=v_n$. Otherwise, consider the ordering obtained by swapping $v_k$ and $v_{k+1}$. Since for $i$ which are neither $k$ nor $k+1$ the set $\{v_j|j<i\}$ does not change when we make this swap, this new ordering contradicts our choice unless $v_k$ is adjacent to $v_{k+1}$ and k+1 is at least 3. But this is impossible if k exceeds 2, as $v=v_k$ has only 2 neighbours and if k>2 then by our choice of ordering both these neighbours appear before it in the ordering and so neither is $v_{k+1}$. So k must be 2. But, again, swapping $v_k$ and $v_{k+1}$ contradicts our choice of order, because v is adjacent to both $v_1$ and $v_3$. Thus, v must be $v_n$. Restricting the ordering to G-v, we see that it shows that G-v is a 2-tree. QED.

It follows that we can recursively test if G is a 2-tree and if so find a rooted tree (T,r) and sets of triangles which show it is a tree as follows:

If G is a triangle then T is a tree with one node r, and we arbitrarily label the vertices of G as $a_r,b_r,c_r$. Otherwise, determine if G has a 2-leaf. If not then return that G is not a 2-tree, otherwise find a 2-leaf v and recurse on G-v. If G-v is not a 2-tree neither is G, return this fact. Otherwise take the rooted tree and set of triangles which shows that G-v is a 2-tree, find a node p of T such that the triangle corresponding to p contains the edge of G-v whose endpoints are the neighbours of v in G, add a leaf s incident to p to obtain a new tree from the tree for G-v,set $a_s$=v, choose one neighbour of v to be $b_s$ and the other to be $c_s$. Return the new rooted tree and set of triangles.

Since testing if a vertex is a 2-leaf can be done in O(|E(G)|) time, this algorithm takes O(|V(G)||E(G)|) time. I asked you to see if you could speed it up.

Given a graph G and associated tree T with a triangle of G for each node of T, for each subtree S of T, we let $G_S$ be the subtree of G obtained by taking the union of the triangles corresponding to nodes in S, and we let $V_S$ be the union of the vertices in these triangles. Thus, $V_S=V(G_S)$.

We also noted that given a 2-tree G, and a rooted tree (T,r) with an associated triangle Tri(x) of G for each node x of the tree so that every edge of G and every vertex of G is in one of these triangles, we can choose a root and label the vertices of each Tri(s) as $a_s,b_s,c_s$ as in the definition of 2-tree precisely if for every edge st of T, letting $T_s$(respectively $T_t$) be the component of T-st containing s (respectively t), we have that Tri(s) ∩ Tri(t) is an edge, and there are no edges between the disjoint sets $V_s$- (Tri(s) ∩ Tri(t)) and $V_t$- Tri(s) ∩ Tri(t).

We also discussed dynamic programming on trees and 2-trees:

To begin, we presented an algorithm to find the largest stable set in the tree. As discussed on pages 560-562 of Kleinberg and Tardos, essentially exactly the same algorithm also finds the maximum weight stable set in a tree. For 2-trees we can find the maximum weighted stable set given a 2-tree G, and a weight w(v) for each vertx of v using the following 2 phase process:

In the first phase, we find a rooted tree (T.r) and a triangle with vertex set $a_s,b_s,c_s$ for each node s of T as in the definition of 2-tree using the algorithm above. For each node s of T we denote the tree consisting of s and its descendants as T(s). In the second phase, we traverse the nodes of T in post-order computing for each s a table containing for each x in $\{a_s,b_s,c_s,\emptyset\}$, the maximum weight w(s,x) of a stable set in $G_{T(s)}$ whose intersection with $\{a_s,b_s,c_s\}$ is x. We return max ($w(r,a_r\}$, $w(r,b_r\}$, $w(r,c_r\}$, $w(r,\emptyset\}$).

If s is a leaf then we set $w(s,\emptyset) = 0$, and for each x in $\{a_s,b_s,c_s\}$, we set w(s,x)=w(x).

Otherwise, we let c(1),…,c(k) be the children of s. We let $x_i=a_{c(i)}$ be the unique vertex in the triangle Tri(s) corresponding to $c_i$ which is not in the triangle corresponding to s. When we arrive at s we have computed our tables for the children of s, so we can compute the table values for s as follows.

We set: $w(s,\emptyset) = \sum_1^k \max\,(w(c_i\,,x_i), w(c_i,\emptyset))$.

For each x in $\{a_s,b_s,c_s\}$, we set:

$$w(s,x) = w(x) + \sum_{i\in\{1,..,k\}\ s.t.\ x\notin Tri(s)} \max\,(w(c_i,x_i), w(c_i,\emptyset)) + \sum_{i\in\{1,..,k\}\ s.t.\ x\in Tri(s)} w(c_i\,,x) - w(x)$$

Following the analysis in Kleinberg and Tardos, the second phase takes time linear in the number of nodes of T and hence linear in the number of vertices of G.