Lingfu Zhang February 10, 2015

6.046 Problem 1-1

Collaborators: *none*

(a) Consider the situation of $V = u_0, u_1, u_2$ and $E = (u_0, u_1), (u_1, u_2)$, with $p_0 = 2$, $p_1 = 3$, $p_2 = 2$. Using the "greedy" algorithm described in the problem, we will choose u_1 at the first step, and remove u_1, u_0 and u_2, as u_0 and u_2 are neighbors of u_1. Then the total profit we get is 3. However, if we select u_0 and u_2 instead, we can get a total profit of 4, which indicates that the "greedy" algorithm doesn't work.

(b) This problem is that given a tree every vertex of whom has a weight related to it, we need to find a set of vertices with maximum total weight and no two of selected vertices are adjacent.
Let's randomly select a vertex u_0 as the root of the tree. For a given vertex u_i, there is exactly one path from u_i to u_0.

Then give some definitions:
Let's call the next vertex on the path from u_i to u_0 the "Father" of u_i. Then every vertex in V, except for u_0 itself, has a unique "Father".
For two vertices u_a and u_b, if u_a is the "Father" of u_b, we call u_b a "Child" of u_a. Then any adjacent vertex of u_i is either its "Father" or "Child". (Otherwise the paths from both vertices to u_0 don't contain each other, and we can get a circle.)
Let G_i be a subgraph of G, such that G_i consists of all u_j that u_i is on the path from u_j to u_0, and all edges among these vertices. Then $G_0 = G$, and for any i, $u_i \in G_i$. G_i contains all children of u_i.
Let $A[i]$ be the maximum total weight of G_i (with no two adjacent vertices selected), and $N[i]$ be the maximum total weight of G_i while u_i itself is not selected (with no two adjacent vertices selected). We now have n subproblems (of getting $A[i]$ and $N[i]$). Then we can calculate $A[i]$ and $N[i]$ by:

$$N[i] = \sum_{u_j \ is \ child \ of u_i} A[j]$$

$$A[i] = \max\left(p_i + \sum_{u_j \ is \ child \ of \ u_i} N[j],\ N[i]\right)$$

The reason of doing this is that any G_i can be devided into many G_j and u_i, where u_j are all children of u_i (u_i might also have no child at all). When calculating $N[i]$, u_i itself is not selected, so we have the freedom of selecting the children of u_i. As all the G_j do not influence

each other (there are no edges connecting them), we know that $N[i]$ is simply the sum of all $A[j]$.
When calculating $A[i]$, there are mainly two situations: u_i is selected or not. If u_i is not selected, the result is simply $N[i]$; if u_i is selected, all its children cannot be selected, then the maximum total weight should be the sum of all $N[j]$ adds p_i. $A[i]$ should be the maximum value of these two.
We shall initialize a list $Father[i]$ to all -1. Then define a function "calculateValues(i)", which first get the list of all adjacent vertices of u_i. If the list has no element other than $Father[i]$, let $A[i] = p_i$ and $N[i] = 0$, then return; else, for every u_j in the list, if $j \neq Father[i]$: let $Father[j] = i$, and do "calculateValues(j)". Finally, calculate $A[i]$ and $N[i]$ using all $A[j]$ and $N[j]$.
We directly call "calculateValues(0)". Then it will call "calculateValues" for all children of u_i, then the all grandchildren of u_i, etc. As this graph is connected, all vertices will be called, and exactly once, because every vertex, except for u_0, has exactly one Father.
For every subproblem, (every calling of "calculateValues"), it does some addings and a comparing. However, as every vertex has one Father, every $A[i]$ and $N[i]$ is added exactly once. The total running time of the above process is $\Theta(n)$.

Now we have all $A[i]$ and $N[i]$, the next step is to find the list "selectedVertices". This can be done by running a "check(i)" function: first compare $A[i]$ and $N[i]$. If $A[i]$ is greater than $N[i]$, add i to "selectedVertices" and check all its grand children; else, check all its children.
We can prove that after running "check(i)", total weight of selected vertices in G_i is $A[i]$. We can prove this by induction. For a vertex u_i without any child, running "check(i)" adds itself to "slectedVertices" list, which makes sure that $A[i]$ is reached. For any u_i, if $A[i]$ is greater than $N[i]$, we know that $A[i]$ is calculated from

$$A[i] = p_i + \sum_{u_j is child of u_i} N[j]$$

By running "check" on every grandchild u_k of u_i, $A[k]$ is reached in G_k for all u_k. Then for every child u_j of u_i, the total weight of selected vertices in G_j is $N[j]$. Then the total weoght of selected vertices in G_i is $A[i]$.
On the other side, if $A[i] = N[i]$, we know that

$$A[i] = N[i] = \sum_{u_j is child of u_i} A[j]$$

By running "check" on every child u_j of u_i, the total weight of selected vertices in G_j is $A[j]$, and then the total weoght of selected vertices in G_i is $A[i]$.
This step run "check" at most n times, and there is only one comparing every running. Thus the running time is also $\Theta(n)$. Then the total running time is $\Theta(n)$.

(c) This problem is basicly the same as (b), except that all weights are equal.

We define a function "find(u_i)" as following:
First, initialize a list L, with only u_i in it. Every time, if the elements in L are l_1, l_2, ... , l_k, and l_k has at least one neighbor different from l_{k-1}, add one of l_k's neighbors (not l_{k-1}) to L; if the only neighbor of l_k is l_{k-1}, add l_k to U, then delete l_k, l_{k-1} and all edges from these two vertices. Then we run "find" for every neighbor of l_{k-1} (not include l_k and l_{k-2}), which will add some other vertices to U. If there is only one vertex l_1 in L and it has no neighbor, add it to U and quit the loop; if there is no vertex in L, quit the loop. Calling "find" from any vertex gives the desired U.
Let's prove that this algorithm works, (namely, gives the maximum number of vertices in U). Let's induce by the number of vertices. When there is only one vertex in G, it's obvious. For n vertices, consider the first time when we have l_1, ..., l_k in L and l_k has only one neighbor l_{k-1}. Deleting l_{k-1} and l_k, and all edges conneting them as well, we get many trees, and each of them has a vertex that used to be l_{k-1}'s neighbor. Then calling "find" on the original graph equals calling "find" on these trees seperately (for a tree doesn't contain l_1, call "find" with the vertex that was l_{k-1}'s neighbor before; for the tree contains l_1, call "find(l_1)"). As every calling gives the desired U for every subproblem, and at most one of u_k and u_{k-1} belongs to U, the total U we get is the maximum.
For the time of running: all we do in this process is adding and removing vertices from L, and deleting edges. Every vertex can be only added and removed exaclty once, as removing the vertex will delete that vertex from further consideration. As what we have is a tree, there are exactly $n-1$ edges. Then the total running time is $\Theta(n)$.

(d) Your solution to Problem 1-1 goes here. Remember, *each problem* should be in a separate LaTeX file so that you can generate one PDF per problem to submit to Stellar.