Find the weight of the minimum weight dominating set of size at least k, in a tree.

First, we define an algorithm to find the mininum weight dominating set of size exactly k.

Define relations:

$$\mathrm{OPT}_0(r,k) = \min \begin{cases} A + w(r) \\ \min_{v \in N(r)} B \end{cases}$$

$$\mathrm{OPT}_1(r,k) = C + w(r)$$

$$\mathrm{OPT}_2(r,k) = \min \begin{cases} D + w(r) \\ E \end{cases}$$

$$n = |N(r)|$$

$$A = \min_{k_1,\ldots k_n} \left[\sum_{x=1}^{n} \mathrm{OPT}_2(N(r)_x, k_x) \right], \sum_{i=1}^{n} k_i = k - 1$$

$$B = \min_{k_1,\ldots k_n} \left[\mathrm{OPT}_1(v, k_1) + \sum_{x=2}^{n} \mathrm{OPT}_0((N(r) \setminus \{v\})_x, k_x) \right], \sum_{i=1}^{n} k_i = k$$

$$C = \min_{k_1,\ldots,k_n} \left[\sum_{x=1}^{n} \mathrm{OPT}_2(N(r)_x, k_x) \right], \sum_{i=1}^{n} k_i = k - 1$$

$$D = \min_{k_1,\ldots,k_n} \left[\sum_{x=1}^{n} \mathrm{OPT}_2(N(r)_x, k_x) \right], \sum_{i=1}^{n} k_i = k - 1$$

$$E = \min_{k_1,\ldots,k_n} \left[\sum_{x=1}^{n} \mathrm{OPT}_0(N(r)_x, k_x) \right], \sum_{i=1}^{n} k_i = k$$

The OPT relations are the same, conceptually, as the ones defined in the lecture slides on the algorithm to find minimum dominating set in a tree.

$\mathrm{OPT}_0(r,k)$ is the minimum dominating set of size k in the subtree rooted at r.

$\mathrm{OPT}_1(r,k)$ is the minimum dominating set which includes r of size k in the subtree rooted at r.

$\mathrm{OPT}_2(r,k)$ is the minimum weight set of size k in the subtree rooted at r which dominates all nodes in the set of nodes not including r itself.

A through E are the updated, more general forms of the sums of the optimal solutions for the child nodes. Instead of merely taking the sum of the lowest weight solutions for the child nodes, we must choose between many different ways to split the remaining number of selectable nodes between the different subtrees. This is done by selecting the set $k_1, ..., k_n$ that minimizes the value of the sum of weights, subject to the constraint that the sum of these k_is must be $k-1$ or k, depending on whether the root node is included in the particular case.

Note that the values of these expressions A to E can be computed by the algorithm from problem 1. Input the target sum of the k_is as the target sum k from problem 1. For the set of T functions, there will be a T for each neighboring vertex and for a neighboring vertex v, $T_v(x)$ will be $\mathrm{OPT}_t(v,x)$, with t being chosen depending on which OPT is being used for v in one of A to E.

To implement this as a dynamic programming solution, we have to examine the dependency graph. When looking at the vertex parameter, since any vertex only depends on OPT values of its children, we can compute the sets of values for the vertices in post-order, which ensures that any vertex will have the OPT values of its children available when it is being computed.

For the k parameter, since for any $\mathrm{OPT}_a(v,b)$ is never dependent on $\mathrm{OPT}_c(v,d)$ for any (a,b,c,d), but only on values less than v, the entries of a row corresponding to any vertex can be computed in any order.

The base cases for the recurrence are the leaves of the tree. For all leaves v,

$$\mathrm{OPT}_0(v,0) = \mathrm{OPT}_1(v,0) = \infty$$

$$\mathrm{OPT}_0(v,1) = \mathrm{OPT}_1(v,1) = w(v)$$

$$\mathrm{OPT}_0(v,x>1) = \mathrm{OPT}_1(v,x>1) = \infty$$

$$\mathrm{OPT}_2(v,0) = 0$$

$$\mathrm{OPT}_2(v,1) = w(v)$$

$$\mathrm{OPT}_2(v,x>1) = \infty$$

The solution to the minimum dominating set of size exactly k is $\mathrm{OPT}_0(r,k)$, where r is the root node.

Time complexity analysis:

First, we compute a post-order traversal order for the tree, which takes $O(n)$ time.

Next, we fill out 3 n by k tables, where $n = |V|$. For each set of 3 corresponding entries in the three OPT tables, we perform the algorithm from problem 1. For A, C, D, and E, this is done once per vertex. For B, this is done $|N(v)|$ times per vertex v. However, there are only $n-1$ total neighbors, so the total number of times B computed is also only $O(nk)$.

The problem 1 algorithm runs in $O(xy^2)$ time, where x is the size number of T functions and y is the target sum. for any time we run the algorithm, the input size is $x = N(v), y = k_0$, where $N(v) \in O(n)$ and $k_0 \leq k$, so any instance of the problem 1 algorithm used here will be $O(nk^2)$ and therefore the overall runtime for this section is $O(nk * (nk^2)) = O(n^2k^3)$.

We then repeat the entire algorithm for target sum values of k up to n, or $n-k$ times, to find the minimum weight dominating set of size at least k. This yields a total runtime of $O(n^2k^3(n-k))$.