Lecture 5: Tree Dynamic Programming

Objectives of this lecture

- Quiz 1 summary
- Introduce Tree Dynamic Programs
- Some recaps of trees
- Combining tree DPs with other DPs

1 Philosophy

This lecture is about dynamic programming on trees. It's the last topic in dynamic programming for two reasons:

1. almost any kind of dynamic program be combined with trees, because tree nodes present natural combine points for transitions,
2. many data structures can be viewed as gradually updating dynamic programming solutions on an appropriately chosen tree.

In a rooted tree, each node i has a non-empty list of children, $\textsc{Children}(i)$, and the parent of node $j \in \textsc{Children}(\text{I})$ is $\textsc{Parent}(j)$. The main idea is of tree dynamic programming is to define states corresponding to subtrees. Then comptue $DP[i]$ by combining $DP[j]$ for all $j \in \textsc{Children}(i)$.

2 Example Problems

Quantities that you have seen computation by recursion, e.g. depth and heights of nodes, can be computed in the same asymptotic complexity using tree dynamic programs.

Problem 1: Height Computation

The height of a node is the farthest (measured in number of edges) node from it in its subtree.

In a rooted tree, compute the heights of all the vertices in $O(n)$ time.

To solve this, observe that the height of a node is the max among its children of their height plus the distance to this node. Formally, we define the DP state as

$$\textsc{MaxDown}[i] = \text{length of longest path starting at } i \text{ into the subtree rooted at } i.$$

The transition is an enumeration among i's children, since the path must go into one of them:

$$\textsc{MaxDown}[i] = \max_{j \in \textsc{Children}(i)} \big(1 + \textsc{MaxDown}\big[j\big]\big).$$

Note that to compute this, we must have answers for everthing in $\textsc{Children}[i]$ before computing on it. This can be done in two ways:

1. Turn the tree into a DAG: arrange the nodes in some order that respects depth, either BFS traversal, or sort.
2. Do depth first search, and return DP values as one go up it.

The base case is when i is a leaf (aka. has no children). Here $\textsc{MaxDown}[i] = 0$ since there are no paths starting from i anymore. Alternatively, we can set $\textsc{MaxDown}[i] \geq 0$ for all i, since i itself is in the subtree rooted at i.

Sometimes just 'upward' states are not enough. This happens naturally when the tree is no longer rooted.

Problem 2: farthest point of all nodes in a tree

Given a general tree as a list of its edges.

Compute in $O(n)$ time the farthest node from each node.

Note that when given a general tree, we can still root it arbitrarily. Both BFS and DFS 'direct' the edges so that each node has a parent pointer.

Once the tree is rooted, we first compute the max length of path into its subtree in the same way above.

However, the issue is that the maximum length path from a node i may go upwards: e.g. if we root a path at one end point, anything past the half way point will need to go upwards.

We address this by adding another state

$$\textsc{MaxUp}[i] = \text{the max length of a path to } i \text{ that goes through } \textsc{Parent}(i).$$

The transition on this is that a path upwardsa from i either goes further up from the parent of i, in which case it is

$$1 + \textsc{MaxUp}[\textsc{Parent}(i)]$$

or it takes a turn downward at $\textsc{Parent}(i)$, at which point we need to be careful to make sure it doesn't go back into i:

$$2 + \max_{j \in \textsc{Children}(\textsc{Parent}(i)), j \neq i} \textsc{MaxDown}\big[j\big].$$

Combining these gives the transition

$$\textsc{MaxUp}[i] \;=\; 1 + \max\left\{0, \textsc{MaxUp}[\textsc{Parent}(i)], \max_{j \in \textsc{Children}(\textsc{Parent}(i)), j \neq i} \left(1 + \textsc{MaxDown}\left[j\right]\right)\right\}.$$

The base case is that since the root has no parent edge,

Note that on trees with arbitrary degree, this may not be fast: computing this for each $i \in \textsc{Children}[k]$ requires going through the listof all chidren. So solving this for all i that are children of some node k can take up to $|\textsc{Children}(k)|^2$ time. This is problematic when degrees are large, e.g. the star graph on n vertices.

To make this faster, note that we can just compute the smallest two numbers of

$$\left\{\textsc{MaxDown}\left[j\right] \mid j \in \textsc{Children}(k)\right\},$$

and casework based on whether i is the child that gives the minimizer.

Almost any dynamic program can be combined with a tree. Here is an example motivated by knapsack.

Problem 3: max weighted unrelated subset of size k

A node i in a tree is an ancestor of a node j (and j a descendant of i) if j can reach i by taking parent pointers.

Given a tree with n nodes, each with weigth $w_1 \ldots w_n$, and a number k, find the maximum weight of a set of k nodes such that none are ancestors/descendants of each other.

Note that if a subtree has nothing above, then we're just solving the same problem inside it for some number of selected nodes that we can guess, as part of the state. This leads to the DP state of

$$\textsc{MaxValue}[i][x],$$

the max value that can be achieved if we select x nodes from the subtree at i. The answer is then $\textsc{MaxValue}[root][k]$.

The base cases are:

1. $\textsc{MaxValue}[i][0] = 0$: selecting no nodes,
2. $\textsc{MaxValue}[i][1] \geq w_i$: selecting i, which means nothing more in its subtree can be selected. Note however that we can't set it to $=$ here: it's possible to pick one node in i's subtree with bigger weight.

The transition occurs when we do not select i. In which case, we need to distribute k among its children.

We start with the easier case where i only has two children (the 1 child case is a direct transition).

Let $\textsc{Children}(i) = \{j_1, j_2\}$. We want to split x nodes into x_1 and x_2, among these two subtrees. This leads to the transition:

$$\textsc{MaxValue}[i][x] = \max_{0 \le x_1, x_2, x_1 + x_2 = x} \textsc{MaxValue}\left[j_1\right][x_1] + \textsc{MaxValue}\left[j_2\right][x_2].$$

The cost of this transition is $O(k)$. Because there are nk states, the total cost of this is $O(nk^2)$.

note: the second base case of $\textsc{MaxValue}[i][1] \ge w_i$ can also be viewed as an extra transition for the case where $k = 1$. Richard likes to put it here because it's not applicable in the general case: as soon as i is selected, nothing in its subtree can be selected. It is completely ok to incorporate it as an extra transition instead of as a base case.

The more general setting of trees with arbitrary degree/number of children actually reduces to the degree 2 case. There are two ways to think about this, whose implementations are actually very similar.

1. establish DP states for each prefix of the children (pick k from the first i children of p),
2. or just modify the tree to turn high degree node into paths, and put $-\infty$ on the newly created nodes to ensure that they will never be picked.