DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

In this and next lecture I want to talk about dynamic programming on trees. It’s the last topic in dynamic programming for two reasons:

1. almost any kind of dynamic program can be carried over to trees, and

2. many advanced data structures can be viewed as gradually updating dynamic programming solutions on an appropriately chosen tree.

Recall a tree is a connected graph with no cycles. We can also root the tree: aside from a root node which we will denote using \( r \), every other node has a unique parent. This also means that each node has zero or more children.

The basic idea is of tree based dynamic programming is to define states corresponding to subtrees. Then the states of \( p \)'s children, \( q \in \text{CHILDREN}(p) \), are combined, to form the solution for \( p \) as well.

**Problem 5.1.** In a rooted, weighted, undirected tree, compute the height of all the vertices in \( O(n) \) time. The height of a node is the farthest node from it in the subtree below it.

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 \( \text{MaxBelow}[p] \) as the DP state.

The base case is \( \text{MaxBelow}[p] \geq 0 \), since \( p \) is in the subtree rooted at it.

The transition is an enumeration among \( p \)'s children:

\[
\text{MaxBelow}[p] = \max_{q \in \text{CHILDREN}(p)} w_{pq} + \text{MaxBelow}[q].
\]

Note that to compute this, we must have answers for everything in \( \text{CHILDREN}[p] \) before computing on it. This can be done in two ways:

1. Arranging 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.

I used to like to do things by BFS order, because I can go explicitly manage memory when the nodes have additional states (which can sometimes make things 10 times faster). Eventually I went over to the other method of doing depth first search because it’s easier to code, although I’m still not sure by how much.

One can imagine all sorts of variant of such aggregations, including:
1. Count the number of paths that are at maximum length.

2. Have edges of two different colors 1 and 2, and only consider longest path below that uses at most $k$ edges of color 2. This requires augmenting the nodes with another dimension of size $k$.

3. Computing the diameter of a tree: this can be done by considering the top-most vertex in some path. That is, when handling $p$, we consider the two children with the first and second max heights, and check whether their sum exceeds diameter.

Sometimes just ‘upward’ states are not enough. Simplest example of this is one where we also need to track the max downward length in order to figure out the max distance one can reach from each node.

**Problem 5.2.** Compute in $O(n)$ time the farthest node from each node of a tree.

This problem is available at [https://dmoj.ca/problem/dmopc14c4p6](https://dmoj.ca/problem/dmopc14c4p6)

This requires us computing $\text{MaxDown}$, the max length of a path through $p$’s parent. The transition is:

$$
\text{MaxDown}[p] = w_{p, \text{Parent}[p]} + \max \left\{ \text{MaxDown}[^{\text{Parent}[p]}], \\
\max_{q \in \text{Children}[^{\text{Parent}[p]}]} \left( w_{q, \text{Parent}[p]} + \text{MaxUp}[q] \right) \right\}.
$$

Note the case of $q$ coming from a sibling of $p$ requires the path to go up to $\text{Parent}[p]$ first, so it’s coming up from $q$, and not downward.

Naively this runs in time quadratic in the degrees of vertices, but we can propagate things downward to all children of some node $r$ in $O(\text{degree}[r])$ time by computing, for each child of $r$,

$$s \in \text{Children}[r]$$

the max value of $w_{rt} + \text{MaxUp}[t]$ for $t \neq s$ in $O(\text{degree}[r])$ time total.

============== stopped here on Monday ==================

**Problem 5.3.** Given a tree where some edges can be used once, some twice, find the longest walk.

This problem is available at [https://dmoj.ca/problem/dmopc18c4p5](https://dmoj.ca/problem/dmopc18c4p5)

This problem can be approached via some kind of case work based on whether paths start/end in subtrees This is given in [https://dmoj.ca/problem/dmopc18c4p5/editorial](https://dmoj.ca/problem/dmopc18c4p5/editorial)

On the other hand, the connectivity DP things discussed last week enables us to take a more global view.

Specifically, we want to select some subset of edges so that:

1. the edges are connected.
2. 2 vertices have odd degree, rest have even degrees.

Note that connectivity in tree is easy to enforce, it’s just ‘no longer consider children that we don’t put edges to’.

So the DP states just track:

1. parity of number of times edge to parent is used.

2. number of odd degree nodes allowed in this subtree.

And the transitions enumerate the number of times that the edges to children are used, and the number of degree 1 nodes used in them. For the second part, we need to take into account whether the degree of this node itself ends up odd, but that’s just one extra if statement.

The number of states is $O(1)$ per node (parity of edge to parent, number of odd degree nodes in sub-tree), so the total running time is $O(n)$.

Note that the ‘number of odd degree vertices’ is essentially combining tree based dynamic programming with knapsack. This can be taken further. Such problems initially plays out like knapsack on trees, but there are a few more tree-specific tricks coming in as well.

**Problem 5.4.** Given a tree with $n$ weighted nodes, 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

$$\text{MaxValue}[p][x],$$

the max value that can be achieved if we select $x$ nodes from $p$’s subtree. The base case are $\text{MaxValue}[p][0] = 0$ and $\text{MaxValue}[p][1] \geq \text{weight}_p$.

The transition occurs when we do not select $p$. In which case, we need to distribute $k$ among its children. This is actually another knapsack problem that we can reduce to the 2-way split case by either:

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.

These two are actually more or less the same implementation-wise: each new node in the second approach corresponds to exactly a prefix of the children of $p$. However, when analyzing the algorithm, it’s much easier to only work with degree 2 nodes, so let’s do that first.
Let the two children of \( p \) be \( q_1 \) and \( q_2 \). We want to split \( x \) nodes into \( x_1 \) and \( x_2 \), among these two subtrees. This leads to the transition:

\[
\text{MaxValue}[p][x] = \max_{0 \leq x_1, x_2, x_1 + x_2 = x} \text{MaxValue}[q_1][x_1] + \text{MaxValue}[q_2][x_2].
\]

The cost of this transition is \( O(t) \leq O(k) \). Because there are \( nk \) states, the total cost of this appears to be \( O(nk^2) \) at a first glance.

The cost is actually significnatly less. First note that it’s actually bounded by the smaller size of a child subtree:

\[
\min\{k, \text{Size}[p]\} \cdot \min\{k, \text{Size}[q_1], \text{Size}[q_2]\}.
\]

We can actually prove that this total cost in a tree is smaller.

**Lemma 5.5.** In a binary tree with \( n \) nodes, for any value \( k \), the total of the value in Equation 1 summed over all nodes is \( O(nk) \).

**Proof.** We will prove this in two steps, we first handle the case where \( \text{Size}[p] \leq k \). This implies that the size of both children of \( p \) are also at most \( k \) as well.

Note that whenever \( x = x_1 + x_2 \) and \( x_1 \leq x_2 \), we get via \( x_1 \leq x/2 \leq x_2 \):

\[
x_1^2 + x_2^2 + x_1 \cdot x \leq x_1^2 + x_2^2 + 2x_1x_2 = (x_1 + x_2)^2 = x^2.
\]

for any \( y \leq x/2 \). So we get that as long as \( \text{Size}[p] \leq O(k) \), the cost is at most \( O(k \cdot \text{Size}[p]) \). So we can prove by induction on the value \( x \) that a node \( x \) incurs a cost of at most \( x^2 \).

For the case where the size of \( p \) is more than \( k \), observe that there are at most \( O(n/k) \) nodes with both children having size at most \( k \). So the total cost among such nodes is at least

\[
O\left(\frac{n}{k}\right) \cdot O\left(k^2\right) = O\left(nk\right).
\]

Then the remaining case is that one of the children has size \( < k \), but \( p \) has size more than \( k \). For this case, observe that we can charge a cost of \( k \) to all nodes in the smaller subtree. Such nodes are never charged again, because in that subtree there are no more nodes of size more than \( k \). So once again we get a contribution of \( O(nk) \).

On additional trick that can be done to this problem is that we can reduce the total memory usage to \( O(\log n \cdot k) \). We always recurse onto the child with bigger size first, and use tail recursion to directly pass up the size \( k \) knapsack table. This ensures that the only things we need to keep ‘on the stack’ are the \( O(\log n) \) ancestors with successively doubling sizes.

In my opinion the most interesting tree DPs are ones where a general graph is solved against a tree.

**Problem 5.6.** Given a tree with \( m \) additional edges with associated costs. Find the max total cost of these edges that can be added to \( T \) so that the resulting graph has no even lengthed cycles.
This problem, with an additional constraint on vertex degrees, is available at [https://dmoj.ca/problem/ioi07p6](https://dmoj.ca/problem/ioi07p6).

We can discard all off-tree edges whose induced cycles along the tree have even length right away.

Then we are left with edges that produce odd lengthed off-tree cycles. If we picked any two such cycles that have overlap, then taking their union, minus the part in common, gives another cycle. This cycle has even total length because each overlapped edge gets subtracted twice. So we’ve reduced the problem to finding the max weight of a subset of cycles that do not overlap.

Then the dynamic programming states are just one per cycle: each cycle has a topmost vertex, and the states corresponds to the max weight of cycles in that subtree that can be picked, subject to this particular cycle being picked as well.

The trickier part is in the transitions: consider the cycle formed by off-tree edge $uv$, whose tree path has topmost vertex $p$. Then none of the other cycles can utilize edges on the path from $u$ to $p$, as well as $v$ to $p$. We can piece these together once again: for each node $q$, we need to compute a table corresponding to the max weight of things that can be picked using any edge in its subtree, except the edge $qr$ for each $r \in \text{Children}[q]$. For some other children $s$, there are several possibilities:

1. the $qs$ edge is not used. Then the max cost is the max cost of $s$’s subtree, which can be computed similar to how we compute $q$.

2. the $qs$ edge is used by some cycle topping out at $q$. Then this pairs together $s_1$ and $s_2$.

Thus, each cycle topping out at $q$ acts like an edge among two of its children, and we want to pick a max weighted subset of these that are vertex-disjoint. There are two ways of solving this:

1. running some kind of max weighted matching algorithm, [https://en.wikipedia.org/wiki/Maximum_weight_matching](https://en.wikipedia.org/wiki/Maximum_weight_matching) or,

2. another dynamic program that takes time $2^{\text{degree}}$, which the problem linked has additional constraints set up toward.