Problem 3: Constructing trees by using a fixed library of tree patterns

We propose an algorithm based on dynamic programming that computes the answer to the problem in $O(nS + nm^2)$ time, using $O(nm^2)$ memory. We define S as the total number of nodes in all the templates.

Fitting templates on the tree

According to the clarifications, the children of a node are numbered, both in templates and in the tree that we have to build. Therefore, the following simple algorithm can check if a template's tree T can be overlaid on a subtree U of the original tree:

- if T is a leaf return *true*
- for each child of T's root, check that U's root has a corresponding child, then use this algorithm recursively to check that the subtree rooted the child of T's root can be overlaid on the subtree rooted at the corresponding child of U's root

If the trees are represented using the *left-child, right-sibling representation* described in Section 10.4 in CLRS, and a node's children are ordered left-to-right in increasing order of their numbers, then the above algorithm can check if a template tree T can be overlaid on a subtree in $O(s_T)$ where s_T is the number of nodes in T. Therefore, we can check which of all the given templates can be overlaid on a subtree in $O(S)$ time, where $S = \sum_T s_T$, as defined in the introductory section.

The algorithm above can be easily augmented so that, in case the template can be overlaid on the subtree, the algorithm returns a list of mappings from the template tree's leaves to the corresponding nodes in the subtree. The modification can be done without changing the asymptotic running time:

- in the base case, return a mapping between T's root and U's root
- when T has children, return the result of merging the lists returned by recursive invocations of the algorithm on each of T's children

Setting up the dynamic programming framework

In this section, we state that the given problem has optimal sub-structure. Based on the proof of this statement, we design the basic dynamic programming algorithm.

Suppose the optimal solution overlays a template T_0 on the root, and the template is bound to templates $T_1, T_2 \ldots T_k$ at nodes $v_1, v_2 \ldots v_k$. The total binding energy is the sum across all i's of the binding energy between T_0 and T_i plus the binding energy in the subtree rooted at v_i. For each i, the templates in the subtree rooted at v_i minimize its contribution to the sum mentioned above. Otherwise,

there would be a choice of templates for the subtree rooted at v_i that would yield a lower total binding energy, which contradicts the assumption that we are considering an optimal solution.

This suggests breaking the problem into sub-problems consisting of computing $\mu(i,j)$, the minimum binding energy achievable at the subtree rooted at v_i by overlapping the root of template (T_j, a_j, c_j) on the subtree's root. We use ∞ to express the impossibility of achieving a solution. The solution to the problem is $\min_j \mu(r,j)$ where v_r is the root of the given tree.

The recurrence relation to compute $\mu(i,j)$ is reasonably simple, but long-winded:

- if T_j can be overlaid on the subtree rooted at v_i:
$$\mu(i,j) = \sum_{v_k \text{ maps to a leaf in } T_j} \min_l (c_j \cdot a_l + \mu(k,l))$$
- else $\mu(i,j) = \infty$

In effect, the algorithm is working its way one node at a time, and attempts to overlay all possible template trees over the subtree rooted at the node it's working on. If the overlay is possible, then the algorithm uses the optimal sub-structure of the problem to compute strategies for overlaying templates on the uncovered nodes. With this insight in mind, it makes sense to set the initial conditions to $\mu(i,j) = 0$ for every leaf v_i and every possible template.

Implementation

Since we are dealing with a tree, it is easy to see that the recurrence is well-defined in light of the initial conditions above: we have values for the tree's leaves, and we have a way to compute $\mu(i,j)$ for every internal node using the values of $\mu(i,j)$ for its children. This also suggests a very easy approach to computing $\mu(i,j)$ bottom-up: if we do a topological sort of the tree, and process nodes in reverse order, we are guaranteed to compute $\mu(i,j)$ for a node before we need it for one of the nodes above it in the tree.

A naïve implementation based on the above recurrent formula would take time proportional to $O(nS + nm^3)$. The first term is owed to the fact that each template tree must be checked for each node, and the second term is obtained by analyzing the formula for $\mu(i,j)$.

However, we can "shave off" a factor of m with a simple observation: $\min_l (c_j \cdot a_l + \mu(k,l))$ does not depend at all on i. So we can break up our main recurrence into the two relationships below:

$$\mu(i,j) = \sum_{v_k \text{ maps to a leaf in } T_j} \nu(k,j)$$
$$\nu(k,j) = \min_l (c_j \cdot a_l + \mu(k,l))$$

We observe that $\mu(i,j)$ only needs values of $\nu(k,j)$ corresponding to descendants of v_i in the tree, so we can still use our bottom-up approach to computing $\mu(i,j)$, with the only modification that once we

compute $\mu(i,j)$ for a node v_i, we also compute $\nu(i,j)$. A straight-forward algorithm that computes $\mu(i,j)$ according to the formulas and indications above will have a running time of $O(nS + nm^2)$ and use $O(nm)$ memory.

Reconstructing the solution

Once we know the binding energy of the optimal overlay (assuming the tree can be covered with the given templates), we are also interested in producing the set of templates corresponding to the overlay.

Like the other dynamic problems we have studies, this problem has a simple solution recovery strategy, assuming we store a bit of extra information:

$$\rho(k,j) = \operatorname*{argmin}_l(\, c_j \cdot a_l + \mu(k,l))$$

In effect, when we compute $\nu(k,j)$, we also keep track of the template associated with the minimum value in $\nu(k,j)$.

The template we use to cover the tree's root is $\operatorname{argmin}_j \mu(r,j)$ - a slight modification of the formula we use to find the minimum overall binding energy. If we know the template we will use to cover the root of a subtree, we can use the following algorithm to compute a covering for the entire subtree:

- if the root of our subtree has no children, stop
- overlay the template tree we use to cover the root, and obtain the nodes mapped to the leaves of the template tree
- for each node mapped to a leaf in the template tree, use $\rho(i,j)$ to find out the template that will cover the subtree rooted there, then call this algorithm recursively with the information

The recovery algorithm proposed above has a running time of $O(nS)$, since it only needs to do overlay tests in order to compute the list of nodes mapped to leaves in template trees. The correctness of the solution recovery algorithm is rather straight-forward, as the algorithm is similar to all the other solution recovery algorithms we have studies.

Just like with previous dynamic programming problems, memory can be traded off for running time: we can compute the entries of $\rho(k,j)$ on demand instead of storing an array. This saves the memory for ρ at the cost of increasing the solution recovery running time to $O(nS + nm^2)$, which doesn't change the overall asymptotic running time.

Wrapping up

The solution we propose is a straight-forward application of dynamic programming. We have already argued for optimal sub-structure, and sub-problem overlap is reasonably obvious in our formulation. As with most dynamic-programming solutions, the recurrence formulas (specified above) are the core of the algorithm.