1. Independent Set on a Tree

Recall the problem statement for independent set: Given an undirected graph, $G$, find the largest set of vertices, $I \subseteq V$, such that no edges go between two vertices in $I$. In general, this problem is NP-hard. However, on trees it’s very tractable. We’ll come up with an $O(n)$ algorithm for it!

(a) What seems like a reasonable sub-problem?

**Solution:** We’ll solve for the maximum independent set on each subtree.

(b) Assuming we have all of the sub-problems solved, how can we combine them?

**Solution:** We can split into two different cases. If we’re trying to solve for $I(u)$ (that is, the maximum independent set of the tree rooted at $u$), we can either take $u$ or not. If we take $u$, we can’t take any of $u$’s immediate children, we move directly to the grandchildren. So in this case we get

$$1 + \sum_{w \in \text{grandchildren of } u} I(w)$$

And if we don’t take $u$, we can take the children, so then it’s

$$\sum_{w \in \text{children of } u} I(w)$$

Overall, we get

$$\max \left( 1 + \sum_{w \in \text{grandchildren of } u} I(w), \sum_{w \in \text{children of } u} I(w) \right)$$

(c) What’s the runtime of our proposed solution?

**Solution:** Let’s examine how many times we may have to look at each node: We have to look at it when we solve it, and then its parent will look at its solution, and its grandparent will look at its solution. In total, each node is used at most 3 times. The overall complexity, therefore, is $O(n)$
2. Time Warping

Time warping is a very important problem in speech and video processing. We’ll use the example of speech here. Suppose I say “Dynamic programming!” You might also say “Dynamic programming!”, but you might say it a little faster or slower, with a different voice, and with different stress on each syllable. But somehow we want to find out that we said the same thing. Time warping is the problem of finding some mapping from each sound you made to sounds I made. Formally, we want this:

Given a sequence $x$ of $N$ real numbers, a sequence $y$ of $M$ real numbers, and a cost function $c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$, an $(N, M)$-warping path (just warping path if $N$ and $M$ are clear) is a sequence $p$ of length $L$ such that $p_i = (x_i, y_i)$, satisfying the following conditions:

i. $p_1 = (1, 1)$ and $p_L = (N, M)$
ii. $p_{i+1} - p_i \in \{(0, 1), (1, 0), (1, 1)\}$

Clearly $L \leq N + M$.

We can see how this defines an alignment between sequences $x$ and $y$. $x_{p_i}$ aligns with $y_{p_i}$, and our first condition ensures that the start and end of each sequence is aligned. Now we can introduce a notion a time warp’s cost. The cost of sequence $p$ is defined as

$$c_p = \sum_{i=1}^{L} c(x_{p_i}, y_{p_i})$$

So our goal is to find $p$ with minimum $c_p$. To do so, let’s introduce the notion of partial sequences $s(1 : k)$, representing the first $k$ elements of $s$. Now we should be able to build our DP algorithm for the problem.

**Solution:** Our algorithm will construct an $N \times M$ matrix, $D$, where $D_{ij}$ is the optimal cost for a warping path between $x(1 : i)$ and $y(1 : j)$. Clearly

$$D_{1j} = \sum_{k=0}^{j} c(x_1, y_k)$$

$$D_{i1} = \sum_{k=0}^{i} c(x_k, y_1)$$

So our base cases are done.

Now we note that (pardon the notation switch)

$$D(n, m) = \min(D(n - 1, m), D(n, m - 1), D(n - 1, m - 1)) + c(x_n, y_m)$$
This we get by simply casing on the final jump in the sequence. If it’s (1, 0), we go to \( D(n-1, m) \), if it’s (0, 1), we go to \( D(n, m-1) \), if it’s (1, 1) we go to \( D(n-1, m-1) \). Note that we have to end at \((n, m)\) by condition (i). Filling up the \( N \times M \) matrix takes \( O(NM) \) time.

3. **Splitting a Tree** We’ll finish with another NP-hard problem which is easy on trees. For this problem, suppose you’re given a binary tree \( T \), and a natural number \( k \leq |V| \). We want color exactly \( k \) nodes red, and the rest blue, such that the number of edges which go between red and blue nodes is minimized. Using DP, we can find an \( O(n^3) \) algorithm for this.

Let’s first assume that we know the optimal way to color the two children of the root with exactly \( k \) red nodes each. Is this sufficient to compute the overall optimal coloring? Why or why not? If not, what information do you need about the children?

**Solution:** No. We have no clue which nodes to turn back to blue, or even how many need to be reset on each child. Ideally, we’d have solved the problem for all possible values of \( k \) for each subtree, and we’d have solved it on the assumption that the root of each subtree is blue, and the assumption that the root of each subtree is red.

So suppose you had all the information you asked for in part (a). How would you then combine it all to solve the problem?

**Solution:** Suppose we’re trying to find the optimal coloring for tree \( T \) with root \( r \), and children \( A \) and \( B \), with roots \( r_A \) and \( r_B \) respectively, on the assumption that \( r \) is blue. Let this be \( C(T, k, \text{blue}) \). We need the children to cumulatively contribute \( k \) red nodes, so let’s split the red nodes in every possible way between the two children. First, let’s assume that \( A \) gets \( k' \) red nodes, and \( B \) gets \( k - k' \) red nodes. Now we have four problems to solve, because \( r_A \) could be red or blue, and \( r_B \) could be red or blue. Take the solution that, together with \( r \) being blue, gives the minimum (if \( r_A \) is red, add 1 to \( C(A, k', \text{red}) \) because you have a split edge \((r, r_A)\), proceed analogously for \( B \)).

Note that \( k' \) can range from 1 to \( k \), and we do constant work for each value of \( k \). \( k \) is also trivially bounded by \( n \), because we can’t color more nodes red than we have nodes. Thus, given all sub-problems, \( C(T, k, \text{blue}) \) takes \( O(k) \) time to solve. But for each node, we need to solve the problem for all \( k' \in [1 : k] \) (at worst), so each node takes \( O(k^2) \) time to solve. Thus, we take \( O(nk^2) \) time (which is at most \( O(n^3) \)).
This solution is an example of *strengthening*, a technique where we solve a harder problem than what’s given. It’s very useful all over math, especially in DP and induction.