Samuel Dobesh

K Coloring Trees

1. Given a graph $G = (V, E)$ a *k-coloring* of G is a labeling of the vertices with the colors $c_1, c_2, \ldots, c_k$ such that for every edge (u, v) the color of u is different from the color of v. Determining if an arbitrary graph has a k-coloring is NP-Complete. However, it is possible to solve this problem on restricted examples of graphs such as trees.

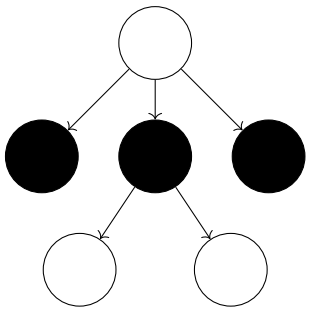
   (a) Give a linear time algorithm that finds a 2-coloring for a tree.

   We know in a valid 2-coloring, every node must be different from it's adjacent nodes. Because trees are defined by only having parent and child relationships, we know that these will be the only adjacent nodes. This means we can find a valid 2-coloring of a tree by simply alternating color at every level. Starting from an arbitrarily colored root, for every node we color it the opposite of it's parent. If at any point we attempt to color an already colored child the other color, the graph has no valid 2-coloring. This could be phrased as a form of BFS, where at every depth of the search the color is swapped. This is the same time complexity as BFS, $O(|V| + |E|)$.

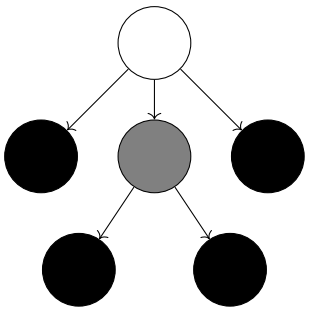
   (b) Think about coloring a tree with two colors so that you have the maximum number of nodes colored c_1. Prove that your algorithm from part (a), with a minor modification probably, can be used to solve this problem. Be precise.

   There are only 2 valid 2-colorings for any given graph, meaning we can find both by finding one and inverting it to get the other. All we have to do to color the maximum nodes c_1 is perform the algorithm from part (a) and keep track of the counts for each color. We can call these variables $count_1$ and $count_2$. Everytime we use a color we incriment the associated counter. If we get to the end, and find that $count_2$ is greater than $count_1$, we can simply iterate through the graph again and swap every color. This would cost an additional BFS, but time complexity wise, $2(|V| + |E|)$ is still $O(|V| + |E|)$.

   (c) Show an example of a tree, T, that can have at most j nodes colored c_1 in a 2-coloring but T can have $j' > j$ nodes colored c_1 in a 3-coloring. Try to find the smallest example of such a tree T.



(i) 2-coloring:j=3



(iii) 3-coloring:j'=4

(d) Give a linear time dynamic programming algorithm that creates a 3-coloring for a tree such that the maximum number of nodes are colored c_1. Justify your answer.

We will start with the simplest case of a leaf node, where there are three possible color configurations. We can notate each configuration with $[c_1, c_2, c_3]$, indicating the color choices composing the subtree. There will always be three possible arrangments, one for each possible coloring of the node. We can call these C_1, C_2, and C_3. For our leaf node, the three possibilities will be $C_1 = [1, 0, 0]$, $C_2 = [0, 1, 0]$, or $C_3 = [0, 0, 1]$. We then work our way up the tree, and at each level we figure out the C_1, C_2, and C_3 that maximize the values for c_1. We can account for neighboring color compatibility by only considering the other two color choices of the children. We want find the sum of all compatible values that maximize c_1, so let's make a helper function, $maxc_1$, which when given two C arrays, returns the one with the maximum c_1. For ease of notation let's also define a set S containing all child vertices of the node we are evaluating.

For each node u, we would then calculate:

$$u.C_1 = [1, 0, 0] + \sum_{v \in S} (maxc_1(v.C_2, v.C_3))$$

$$u.C_2 = [0, 1, 0] + \sum_{v \in S} (maxc_1(v.C_1, v.C_3))$$

$$u.C_3 = [0, 0, 1] + \sum_{v \in S} (maxc_1(v.C_1, v.C_2))$$

Once these values are filled, we can work our way back down the tree from the root. At every stage we choose the configuration, C, with the maximum c_1 that is compatible with its parent node's choice, and fill the node with the color corresponding to that configuration. C_i corresponds to the color c_i.

We can accomplish our algorithm with two tree traversals, the first bottom up traversal filling in color configurations is post-fix, and the second top down is in-fix. This second traversals work is dependent on the branching factor m, as we run $maxc_1$ 3 times on each child. This is constant, but let's include it for fun. We can combine these algorithms as $n + 3mn$, or $(3m + 1)n$, drop constants, and show this algorithm runs in linear time, $O(n)$.