Optimal Tree Labeling

Guilherme Marra & Luiz Pinheiro
Programming Project of INF 421

February 10, 2019

1 Algorithm Conception

During the conception of the algorithm the group realized that the best way to solve the proposed problem was using dynamic programming. Although a possible representation of the acquired data is a graph, since there are vertices and edges that connect them and no pre-defined parentage relationship, this structure would not be practical to solve by dynamic programming. So the group implemented, as suggested by the names used during the proposition of the problem (leafs, optimal tree) a tree, in which the root would be an arbitrary node, as long as it is not a leaf.

1.1 Data Structure

As explained above, the implemented data structure was a tree. Each element the tree is a node with a parent and a linked list of children. Each node in the tree stores:

1. int index: the node index is the sample input index -1, so to work with vectors and matrices the first element of these data structures, with index 0, would have the same index as the first element of the nodes;

2. int label: each letter represents a power of 2, for example A=1, B=2, C=4. So with only a numerical value it can be discovered which are the letters that formed it, for example 7 corresponds only to the label ABC;

3. LinkedList<Node> children : A linked list with all the children of the node;

4. int sizedictionary: saves the number of distinct letters used in the sub-tree representing the node;

5. int[] weightMin: used to save the minimum weight value during the dynamic programming algorithm;

6. int weight: sum of the weight of all links in the subtree whose node is root;

7. Node father: father of the node;

8. int height: height of the subtree whose node is root.

```
1000 public class Node {
     int index;
1002 int label;
     LinkedList<Node> children;
1004 int sizedictionary;
     int[] weightMin;
1006 int weight;
     Node father;
1008 int height;


1010
     public Node(int index, int label) {
1012     super();
         this.index = index;
1014     this.label = label;
         this.children = new LinkedList<>();
1016     this.sizedictionary = 0;
         this.weightMin = new int[52]; // 2*26
1018     this.father = null;
         this.height = 0;
1020     this.weight = 0;
     }
```

1.2 Tree Construction

The tree is built by a function *public Node(BufferedReader bufferedReader, int N, int l)* that returns the root of the built tree. The construction was done by the following algorithm:

1. First with N, number of nodes in the tree, L, number of leaves in the tree and the rest of the entire input in the Buffered Reader, you need to know which nodes are leaves, because they can't be the parent of another node (i.e., they can't be root either). That's why the first N-1 lines of the Buffered Reader with all the links between the nodes were saved in an Array;

2. During the next L rows, all the leaves of the tree were built;

3. By reading the N-1 lines saved in the Array earlier, the tree's kinship links were established. In the first line the root was already defined as the first element that is not a leaf. Even being an arbitrary element, this decision does not interfere in the complexity of the algorithm. In the other lines to know who the parent node would be, the criteria were as follows: if the node is a leaf, it is not the parent; if the node already has a parent, it cannot have another parent; if the node was assigned a root, it has no parent. If the relationship does not fit in any of the three cases the choice of the parent is indifferent. It was considered throughout the construction of the tree that the input is faultless.

2 Algorithm Analysis

The initial idea of the algorithm using dynamic programming was to use recursive function based on the tree property: The weight of a node is the sum of the weight of all its links with its children plus their respective weights. Then, the following distance function was defined, according to the problem:

```
1000    static int distance(int label1, int label2) {
            int ret = 0;
1002        for (int i = 0; i < 26; i++) {
                if (((label1 ^ label2) & 1 << i) > 0) {
1004                ret++;
                }
1006        }
            return ret;
1008    }
```

The main idea then was to divide the problem into all the possible letters and then unite all the solutions. Starting with the letter A, separately, in which vert of the tree if we add A to the label the weight would be minimal. Each individual algorithm then became a minimum binary Hamming-Distance problem, adding or not adding a letter at the vertex.

To solve each letter was used then the algorithm *private static void getOptimalLabel(int l, LinkedList<Node> layer, int LU)* which divides the tree into layers, where layer 0 is the root, camera 1 is its children and the last number layer (root.height-1) is the leaves. In this way, starting from the penultimate camera, it is only checked based on the next camera from which vertices reduce their weight by adding the letter in question. Finally the function *private static void insertOptimalLabel(int l, Node root)* inserts the letters in the given labels.

3 Results

The results with the total weight and time travelled for each code were:

Optimal Tree Labeling		
Test Number	Total Weight	Runtime(ns)
1	24	15181924
2	1682	177523503
3	6936	126763397
4	12927	221278789
5	3360	181343250
6	24971	423605357
7	29937	335196409
8	43443	409882214
9	128297	1506315380
10	214500	1870510630

Due to the speed of the program and the precision in the results, we believe we have made a good resolution of the problem and the order of complexity that any other optimization will be marginal.