Optimal Tree Labeling

ZHAO Tianyuan[*] JIANG Yingjie[†]

January 2019

1 Problem Description

We have a tree $T = (V, E)$, where V is the set of vertices and E is the set of edges. Every vertex $V \in V$ has a label $L(v)$, which is a subset of $\{A, B, ...Z\}$. For an edge $e = (u, v)$, we define the weight $w(e)$ as the Hamming distance between $L(u)$ and $L(v)$, that is:

$$w(e) = |\{X|X \in L(u) \text{ and } X \notin L(v)\} \cup \{X|X \notin L(u) \text{ and } X \in L(v)\}|$$

For example, for an edge $e = (u, v)$ with $L(u) = \{A, B, C, D, E\}$ and $L(v) = \{B, D, E, F\}$, we have $w(e) = |\{A, C, F\}| = 3$.

In the optimal tree labeling problem, we are given an unrooted tree with the labels of all the leaf vertex. The goal is to find some way to label all the non-leaf vertex, such that the total weight of all the edges in the tree is minimized under this labeling.

[*]tianyuan.zhao@polytechnique.edu
[†]yingjie.jiang@polytechnique.edu

2 Algorithm

We define the set of letters as $S = \{A, B, C, ...Z\}$. We have the following lemmes:

Lemma 1. *If we label the tree with a single letter noted i, for an edge $e = (u, v)$, $w(e) = (i \in L(u) \&\& i \notin L(v) || i \notin L(u) \&\& i \in L(v))$*

Corollary 1.1. *The total weight of tree is the sum of weight that we label the tree separately by every $i \in S$.*

Proof. In the case of single letter(i) labeling, We define the label of a node u as

$$L_i(u) = \begin{cases} \{i\} & i \in L(u) \\ \varnothing & i \notin L(u) \end{cases}$$

In the case of multiply letter labeling. The label of any vertices can be written in the form as

$$L(u) = \biguplus_{i=1}^{26} L_i(u)$$

For $e = (u, v)$,

$$\begin{aligned} w(e) &= w(L(u), L(v)) \\ &= w(\biguplus_{i=1}^{26} L_i(u),\ \biguplus_{i=1}^{26} L_i(v)) \\ &= \sum_{i=1}^{26} w(L_i(u),\ L_i(v)) \end{aligned}$$

□

As a result, we can decompose the problem to 26 problems with the same tree structure but only one letter or empty in the label of the leaf vertex. Take the letter i as the general case, the problem is to minimize the total weight with the leaf vertex labeling {i} or ∅.

Lemma 2. *The recursive relationship:*
we note f as a parent node and his children node s, we note $d_i(f)$ as the

weight below node f if $i \in L(f)$, and $d_\varnothing(f)$ if $i \notin L(f)$. For each $i \in S$

$$d_i(f) = \sum_s min\{d_i(s), d_\varnothing(s) + 1\}$$
$$d_\varnothing(f) = \sum_s min\{d_i(s) + 1, d_\varnothing(s)\}$$

First observation of the input format of labeling data shows that if we take the index 1 as the root, the index of the parent nodes will be always smaller than that of children nodes. This characteristic allows us to simplify a lot the calculation of the tree structure.

For each letter we create a table M of 3 dimensions$(26 * N * 3)$ where $M(i, u-1, 0)$ indicates the index of parent node of the node u, $M(i, u-1, 1)$ is the value of $d_i(u)$ and $M(i, u-1, 2)$ is the value of $d_\varnothing(u)$.

Pseudo-code is showed in algorithm 1 and algorithm 2.

3 Analyses of complexity

In the part initialization, we read the input and stock the tree in a table in $\mathcal{O}(N)$, N is the number of vertices.

In the algorithm dynamic programming we begin with the last element in the table and we process the index on by one until node 1, and the time complexity is also $\mathcal{O}(N)$.

So the final time complexity is $\mathcal{O}(N)$ and the space complexity is $\mathcal{O}(N)$.

algorithm 1 Initialization

```
Input: an unrooted tree with the labels of all the leaf vertices. N: number
    of vertices, L: number of leaf vertices.
Output: Minimal total weight of all the edges in the tree.
Initialization: 3-dimensional empty array $(M_i)_i$ of size $26 * N * 3$
for $i = 0$ to $N - 2$ do
    Read index of father node $n_f$ and of child node $n_s$
    for all $l \in \{0, \dots, 25\}$ do
        $M(l, n_s, 0) \leftarrow n_f$
    end for
end for

for $i = 0$ to $L - 1$ do
    Read index of leaf n
    Read the label of leaf n as an array of char: c[ ]
    for all $l \in \{0, \dots, 25\}$ do
        $M(l, n, 1) \leftarrow Max$
        $M(l, n, 2) \leftarrow 0$
    end for

    if c[0]!='$' then
        for $c[i]$ do
            Use ASCII Code to associate 0,...,25 to A,...,Z
            Calculate corresponding letter $l$ using c[i]
            $M(l, n, 1) \leftarrow 0$
            $M(l, n, 2) \leftarrow Max$
        end for
    end if
end for
return $M$
```

algorithm 2 Dynamic Programming

Input: 3-dimensional array M given by the initialization, N number of vertices, L number of leaf vertices

Output: Minimal total weight

```
1: $weight \leftarrow 0$
2: for $l = 0$ to 25 do
3:     for $i = N - 1$ to 0 do
4:         $M(l, M(l, i, 0), 1) += \min(M(l, i, 1), M(l, i, 2) + 1)$
5:         $M(l, M(l, i, 0), 2) += \min(M(l, i, 1) + 1, M(l, i, 2))$
6:     end for
7:     $result += \min(M(l, 0, 1), M(l, 0, 2))$
8: end for
9: return weight
```

4 Results of the algorithm

input sample	N	Optimal solution	Time cost(milliseconds)
labeling.1	100	24	3
labeling.2	2000	1682	15
labeling.3	3000	6936	12
labeling.4	4000	12927	20
labeling.5	5000	3360	20
labeling.6	6000	24971	21
labeling.7	7000	29937	15
labeling.8	10000	43443	15
labeling.9	30000	128297	50
labeling.10	50000	214500	94

The time cost may differ a bit depending on the machine.

5 Generalization

We have mentioned that our algorithm depends on the good format of input data. If vertices are organized disorderly(children node is not all bigger than parent node, or in the N-1 lines of edges, it isn't parent node point to children node). We need to change our algorithm in a more generalized version.

In this case, we use the representation of graph, which means that we use the adjacent list to stock the neighbors of a vertex in the graph. We stock the leaves when reading the input and we begin our dynamic programming from one leaf. The properties of tree tell us that the size of adjacent list of a leaf is 1, so we can execute our dynamic programming like this:

we choose a leaf in the set of leaves, and we compute the distance between this leaf and his single neighbor, i.e. his parent node, and we delete this leaf form the set of leaves and the tree. After, delete the edge between this leaf and his parent node, we verify if his parent node becomes a leaf now, i.e. he has only one neighbor, if yes, we add this parent node into the set of leaves. We continue our processing until there is only one node in the tree, it's the root of this tree.

We need a particular type of queue to stock leaves (*leaves*) that have the function *contains*() which verifies whether a node is a leaf with time complexity $\mathcal{O}(1)$ and the function *add*(*Integer*) which adds a node at the end of the list with time complexity $\mathcal{O}(1)$. The reason that we need to process leaves in order is that we need to guarantee all the "real" leaves (listed in the input file) are treated and it is a root left at last. The reason that we need a $\mathcal{O}(1)$ time complexity *contains* is that there are some leaves undeclared in some input files (like labeling.1.in node 1), we need to identify them and delete them (this won't change the optimal weight)

The generalized algorithm is presented in *algorithm* 3. With the same time complexity of *algorithm* 2, because we just go through all nodes one time.

6 Analyses of complexity for the generalized algorithm

In the implementation of the algorithm, we have used an array of dimension $26 * N * 2$, a tree structure stored in a HashMap whose size will be $\mathcal{O}(2 * (N - 1))$ and a SearchList to store the leaves of the tree which will pass all the nodes of the tree with size N. The final space complexity is $\mathcal{O}(N)$.

Each element in the table is processed only once, time complexity is $\mathcal{O}(26 * N * 2)$. In the SearchList leaves, each node of the tree is added and deleted

algorithm 3 A generalized version of optimal tree labeling, using a representation of graph. $table[i][j-1][0]$: the weight below node j if $i \notin L(j)$, $table[i][j-1][1]$: the weight below node j if $i \in L(j)$. A HashMap ($tree$) that stock all vertices and their neighbors

Input: an unrooted tree with the labels of all the leaf vertices. N: number of vertices, L: number of leaf vertices.

Output: Minimal total weight of all the edges in the tree.

```
1: $HashMap < vertex, neighborvertices > tree$
2: $Queue < vertex > leaves$
3: $int[nb\_letter][N][2]table$
4: Initialization: Read lines and build $tree$ and $leaves$
5: while ! leaves.isEmpty do
6:     $j \leftarrow leaves.poll()$
7:     if leaves.isEmpty() then
8:         root=j; break;
9:     end if
10:    $father \leftarrow the\ only\ neighbor\ of\ j$
11:    for $i = 0$ to $nb\_lettre$ do
12:        $table[i][father-1][0] \leftarrow table[i][father-1][0] + min\{table[i][j-1][0],\ table[i][j-1][1]+1\}$
13:        $table[i][father-1][1] \leftarrow table[i][father-1][1] + min\{table[i][j-1][0]+1,\ table[i][j-1][1]\}$
14:    end for
15:    $tree.remove(j)$
16:    $tree.get(father).remove(j)$
17:    if tree.get(father).size()==1 then
18:        $leaves.add(father)$
19:    end if
20: end while
21: for $i = 0$ to $nb\_lettre$ do
22:     $weight+ = min\{table[i][root-1][0], table[i][root-1][1]\}$
23: end for
24: return $weight$
```

only once, time complexity is $\mathcal{O}(2 * N)$. In the HashMap tree, each node of the tree is added twice and deleted twice,once in the KeySet of HashMap, once in the HashSet, time complexity is $\mathcal{O}(4 * N)$. The final time complexity is $\mathcal{O}(N)$.

7 Results of the algorithm

input sample	N	Optimal solution	Time cost(milliseconds)
labeling.1	100	24	4
labeling.2	2000	1682	19
labeling.3	3000	6936	21
labeling.4	4000	12927	20
labeling.5	5000	3360	16
labeling.6	6000	24971	21
labeling.7	7000	29937	17
labeling.8	10000	43443	23
labeling.9	30000	128297	72
labeling.10	50000	214500	101