INF421
Optimal Tree Labeling

João Marques Andreotti,
Guilherme Vieira Manhaes

31 january, 2023

1 Introduction

For this problem we define $T = (V, E)$ a tree, where V is the set of vertices and E is the set of (undirected) edges. We define $\mathcal{A} = \{A, B, ..., Z\}$ to be the alphabet and $\mathcal{P}(\mathcal{A})$ the power set of the alphabet (any subset of it). Let $L\colon V \to \mathcal{P}(\mathcal{A}) \cup \{\phi\}$ be the labeling function for each vertex, where ϕ represents a vertex that has not yet been labeled (for example all the leaf vertices in the beginning).

Also let $w\colon (u, v) \in E \to d(L(u), L(v)) \in \mathbb{N}$ be the weight function, where d is the Hamming distance between sets.

In this problem, we are given an unrooted tree T and a labeling function that has its outputs defined for all (and only for) the leafs. The goal is to find some way to label the rest of the vertices such that the total cost of all the edges is minimized.

We introduce the following notations for the problem:

1. $\oplus$ is the XOR operator between bits;
2. For any $a \in \mathcal{A}$, let $L_a\colon V \to \{0, 1, \phi, \psi\}$ a labeling function for each letter of the alphabet, where ϕ represents that a vertex that has not yet been labeled and ψ represents an indifferent label for that vertex (see the definition of the algorithm for more details). We initialize, for any $v \in V$:
$$L_a(v) = \begin{cases} \mathbf{1}_{\{a \in L(v)\}} & \text{if } L(v) \in \mathcal{A} \\ \phi & \text{otherwise} \end{cases}$$
3. For any edge $e = (u, v) \in E$ and $a \in \mathcal{A}$, let $w_a(u, v)$ the weight for a letter in the edge:
$$w_a(u, v) = \begin{cases} L_a(u) \oplus L_a(v) & \text{if } u, v \in \{0, 1\} \\ 0 & \text{otherwise} \end{cases}$$
4. For every $v \in V$ and $a \in \mathcal{A}$ we note the number of k-neighbours of v:
$$N_a^v(k) := \sum_{u \text{ such that } (u,v) \in E} \mathbf{1}_{\{L_a(u) = k\}}$$

2 Preliminary Results

We first remark that the labeling problem for the whole alphabet $\mathcal{A}$ can be reduced to $|\mathcal{A}|$ sub-problems considering each letter individually. This stems from the fact that, for $e \in E$, the edge weight can be rewritten as:

$$w(e) = \sum_{a \in \mathcal{A}} w_a(e)$$

And thus, we can naturally express:

$$\min_L \left(\sum_{e \in E} w(e) \right) = \min_L \left(\sum_{e \in E} \sum_{a \in \mathcal{A}} w_a(e) \right) = \sum_{a \in \mathcal{A}} \min_{L_a} \left(\sum_{e \in E} w_a(e) \right)$$

That is to say that deciding whether to include a letter in a node does not depend on the other letters. We can also see that the total cost of the tree can be computed as the sum of costs associated with the tree built for each letter in $\mathcal{A}$.

3 The OTL algorithm

Our proposed algorithm uses a Greedy approach to the problem with a recursive call on the tree.

3.1 Definition

In this part we define the two main procedures used to solve the problem:

Algorithm 1 LabelOptimally

Input: The Tree 'T' and a Labeling 'L'
Output: Total cost of the labeling
Side-Effect: Alters L to represent the optimal labels.

```
1: Find a possible root 'r' for the tree T (any unlabeled vertex will work).
2: Total Cost = 0
3: for a ∈ {A, B, C, ..., Z} do
4:     Total Cost += LabelOneLetter(T, L, r, r, a)
5: end for
6: return Total Cost
```

3.2 Termination

Since the recursive call only goes down the tree (which, by definition, has no cycles), no vertex is visited twice. Every letter of the alphabet is only considered once, so the algorithm terminates.

Algorithm 2 LabelOneLetter

Input: A Tree 'T', a Labeling 'L', the root of the current sub-tree 'root', the root's parent 'parent' and a letter 'letter'.
Output: Total cost of the labeling for one letter.
Side-Effect: Alters L to represent the optimal labels for the chosen letter.

```
if 'root' is a leaf (already labeled) then
    return 0
end if
cost = 0
num_has_letter = 0; num_no_letter = 0
for each child of root do
    cost += LabelOneLetter(T, L, child, root, letter)
    Get child label.
    if child marked "Has Letter" then
        num_has_letter++
    else if child marked "No Letter" then
        num_no_letter++
    end if
end for
if num_no_letter > num_has_letter then
    Mark root as "No Letter".
else if num_no_letter < num_has_letter then
    Mark root as "Has Letter".
else if num_no_letter = num_has_letter then
    Mark root as "Indifferent".
end if
return cost + minimum(num_has_letter, num_no_letter)
```

3.3 Correctness

Consider a vertex (v) whose parent's (p) label is not yet decided and whose children (C_v) all have labels. The cost of the edges associated to this vertex (for a letter $a \in \mathcal{A}$) is thus given by:

$$S_a(v) = w_a((v,p)) + \sum_{c \in C_v} w_a((v,c)) \tag{1}$$

$$= w_a((v,p)) + \mathbb{1}_{\{L_a(v)=1\}} \underbrace{\left(\sum_{c \in C_v} \mathbb{1}_{\{L_a(c)=0\}} \right)}_{N_{0,v}} + \mathbb{1}_{\{L_a(v)=0\}} \underbrace{\left(\sum_{c \in C_v} \mathbb{1}_{\{L_a(c)=1\}} \right)}_{N_{1,v}} \tag{2}$$

Since $|w_a((v,p))| \leq 1$, if $N_{0,v} < N_{1,v}$ (resp. $N_{0,v} > N_{1,v}$) then we must have $L_a(v) = 1$ (resp. $L_a(v) = 0$) in order to minimize the total weight (independent of the parent's label). The only case where we cannot choose a labeling without knowledge of the parent's label is when $N_{0,v} = N_{1,v}$, in which case $S_a(v) = N_{0,v} = N_{1,v}$ because $L_a(v)$ can always later be chosen to match $L_a(p)$ when it is known.

This shows that the decisions made by our proposed algorithm are always optimal, with

indifferent vertices being later chosen to equal their parent's in the tree at no additional cost.

3.4 Runtime complexity

Since every vertex is looked at only once, the complexity of the algorithm is $\mathcal{O}(n)$ in both time and space, where n is the number of vertices.

4 Implementation

Since we require looping through neighboring nodes, the graph was implemented as an adjacency table in *C++*. In order to speed up execution time, the optimal labeling for each letter is computed in a different thread and the results are assembled by the main thread.

The implementation is provided in the repository:
https://github.com/Xlonefy/optimaltreelabeling

5 Benchmarking

All seven provided test cases were evaluated using our algorithm. The results obtained are displayed on Table 1. Additionally the Figure 1 shows that the execution time grows approximately linearly with input size as expected from the $\mathcal{O}(n)$ complexity.

N	Result	Time [ms]
100	24	1.369 ±1.26%
2000	1682	3.095 ±3.19%
3000	6936	3.690 ±3.13%
4000	12927	4.698 ±3.25%
5000	3360	5.855 ±3.82%
6000	24971	6.601 ±3.85%
30000	128297	24.37 ±1.45%

Table 1: Results and run times from test cases.

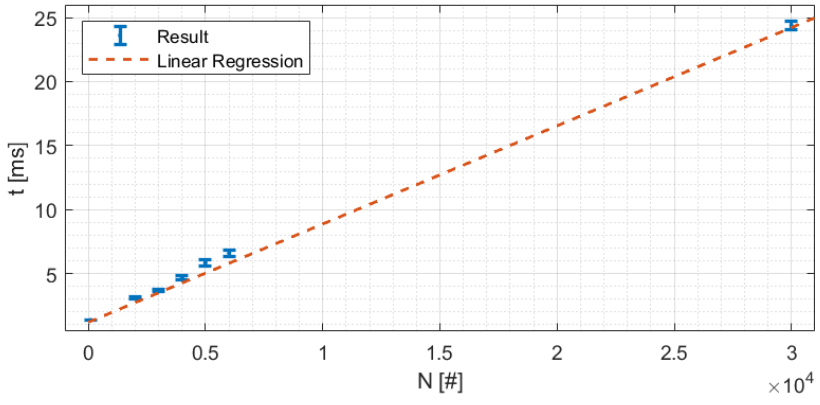

Figure 1: Plot showing execution time versus N.

6 Conclusion

This report shows the termination, correctness and complexity of the proposed algorithm that solves the Optimal Tree Labeling problem, results that are supported by the experimental data. The choice of the data structures, programming language, and other optimizations discussed in the source code also managed to bring the total run-time down significantly.

The two main steps in the conception of this procedure was of realizing that the problem could be broken independently into the letters (which allowed the implementation with multi-threads) and that a greedy approach was possible.