# On an Algorithm of Zemlyachenko for Subtree Isomorphism

Yefim Dinitz [*] Alon Itai Michael Rodeh[†]
Computer Science Department
Technion, Haifa, Israel

March 7, 1999

**Abstract**

Zemlyachenko's linear time algorithm for free tree isomorphism is unique in that it also partitions the set of rooted subtrees of a given rooted tree into isomorphism equivalence classes. Unfortunately, his algorithm is very hard to follow. In this note, we use modern data structures to explain and implement Zemlyachenko's scheme. We give a full description of a free rendition of his method using some of his ideas and adding some new ones; in particular, the usage of the data structures is new.

[*]Dept. of Mathematics and Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. *email*: dinitz@cs.bgu.ac.il.

[†]email: {itai, rodeh}@cs.technion.ac.il

## 1 Introduction

Tree isomorphism is a well known problem which has many applications. A linear time algorithm to solve the problem has been developed by Hopcroft and Tarjan [4], who used it to devise a linear planarity test (see also [1, pp. 84-86]). Several other linear time algorithms were suggested [2, pp.196-199], [5], [6]. Since we shall concentrate on rooted trees, the term tree will refer to rooted trees, unless explicitly stated otherwise.

A natural generalization of the problem of checking whether two trees are isomorphic is that of partitioning a finite set of trees into isomorphism equivalence classes. Of particular interest is the set of rooted subtrees of a given tree. The additional difficulty here is that the rooted subtrees are not disjoint and representing them explicitly may require $\theta(n^2)$ space for a tree of $n$ vertices (consider a path of $n$ vertices).

Clearly, an algorithm for solving the isomorphism equivalence classes of all subtrees of a given tree provides an algorithm for tree isomorphism: To check whether $T^1$ and $T^2$ are isomorphic, construct a tree $T$ whose root has exactly two children, the roots of $T^1$ and $T^2$. The trees $T^1$ and $T^2$ are isomorphic if and only if the subtrees rooted at the roots of $T^1$ and $T^2$ belong to the same equivalence class.

In [8], Zemlyachenko describes a linear time algorithm for tree isomorphism, which partitions the subtrees of a given tree into isomorphism equivalence classes. Unfortunately, his description is terse and incomplete: we were able to follow some of his ideas but not the algorithm itself. In this note, we use modern data structures to explain and implement Zemlyachenko's scheme. We give a full description of a free rendition of his method using some of his ideas and adding some new ones; in particular, our usage of the data structures is new.

A related problem is isomorphism of finite free trees. A *free tree* is a connected undirected graph with no cycles. Any free tree can be converted to a rooted tree by choosing an arbitrary vertex as a root, and directing all edges in direction away from the root. As is well known, free tree isomorphism can easily be reduced to the rooted case, as follows. Consider the longest path in the tree. If it has an odd number of vertices, choose the central vertex as the root. If their number is even, split the central edge $(u, v)$ into two, i.e., add a new vertex $w$ and replace $(u, v)$ by the pair of edges $(u, w)$ and $(w, v)$. Choose $w$ as the root. Now, any two free trees are isomorphic if and only if the resultant rooted trees are isomorphic.

An *ordered tree* is a (rooted) tree in which the children of each vertex are ordered. Isomorphism of ordered trees $T$ and $T'$ is trivial:

> Let $v_1, \ldots, v_d$ be the children of the root of $T$ and $v'_1, \ldots, v'_{d'}$ the children of the root of $T'$.
> $T$ is isomorphic to $T'$ if and only if $d = d'$ and
> for $i = 1, \ldots, d$, the subtree of $T$ rooted at $v_i$ is isomorphic to the subtree of $T'$ rooted at $v'_i$.

The common approach to tree isomorphism is first to convert the trees to ordered trees via a canonical transformation that preserves isomorphism, namely, the original trees are isomorphic if and only if the resultant canonically ordered trees are isomorphic. Thus, every isomorphism equivalence class of trees has a unique ordered tree that serves as the representative of the class. The algorithm suggested in this paper constructs the corresponding canonically ordered tree of a given input tree.

In the process of our algorithm, we assign a distinct integer, called the *canonical index*, to each equivalence class of subtrees of a given tree. We label each vertex $v$ by the index of the equivalence class of the subtree rooted at $v$. Thus two subtrees are isomorphic if and only if their roots are assigned the same index. The algorithm finds all these indexes in linear time. Thereafter, we can use these indexes to determine in constant time whether any two subtrees are isomorphic.

## 2 Subtree Isomorphism

### 2.1 Canonical order

Given a rooted tree $T$ and a vertex $v$ of $T$ other than the root, removing the edge connecting $v$ to its parent, results in two subtrees. Let $T_v$, the *subtree rooted at* $v$, be the subtree that contains $v$. When $T_u$ and $T_v$ are isomorphic, we say that $u$ and $v$ are *subtree isomorphic.*

We start by defining a canonical order, $\leq_C$, on the vertices of a rooted tree $T$. The canonical order $\leq_C$ is defined from the leaves up. The leaves are the minimal elements of $\leq_C$.

Recall that the *height* of a vertex $v$ is the length of the path from $v$ to the furthest leaf of its subtree. To determine whether $u \leq_C v$, first compare their heights. If $height(u) < height(v)$ then $u \leq_C v$. If the heights are equal, we sort the children of $u$ and $v$ to create two lists $u_1 \leq_C u_2 \leq_C \cdots \leq_C u_{d(u)}$ and $v_1 \leq_C v_2 \leq_C \cdots \leq_C v_{d(v)}$. Now, $u \leq_C v$ if and only if the sorted list of $u$'s children is lexicographically less than the sorted list of $v$'s children.

The $\leq_C$ relation induces an equivalence relation, $=_C$, in the natural way: $u =_C v$ if both $u \leq_C v$ and $v \leq_C u$. Thus $u =_C v$ if and only if $height(u) = height(v)$, and $d(u) = d(v)$, and $u_i =_C v_i$ for $i = 1, \ldots, d(u)$. It is easy to see that $u =_C v$ if and only if $u$ and $v$ are subtree isomorphic.

### 2.2 Canonical Index

We extend the $\leq_C$ order to subtree isomorphism classes, i.e., $S \leq_C S'$ if and only if for any $v \in S$ and $v' \in S'$ $v \leq_C v'$.

In order to quickly check whether $u \leq_C v$ we assign to each vertex $w$ a *canonical index* $\xi[w]$. Let $S_1, \ldots, S_m$ be the subtree isomorphism equivalence classes ordered by the canonical order, i.e., $V(T) = \bigcup_{i=1}^{m} S_i$ and $S_1 \leq_C S_2 \leq_C \cdots \leq_C S_m$. The canonical index of a vertex $w$ is the index of the class to which it belongs, i.e., $\xi[w] = j$ if $w \in S_j$.

The indexes are computed from the leaves up. Since the leaves have minimum height and are subtree isomorphic to one another, the class $S_1$ consists of all leaves, i.e., $\xi[v] = 1$ if and only if $v$ is a leaf.

Assume that we have determined the indexes of all the vertices of height less than $h$ and assigned the numbers $1, \ldots, k$. We now consider all vertices of height $h$. For each such vertex $v$ we order its children, $v_1, \ldots, v_d$, by nondecreasing index, i.e., $\xi[v_1] \leq \cdots \leq \xi[v_d]$. We now lexicographically sort the vertices of height $h$ and assign an index to each of them according to this order. If $S_1^h \leq_C S_2^h \leq_C \cdots \leq_C S_{m^h}^h$ is the partition of the vertices of height $h$ into equivalence classes, then for $v \in S_i^h$ we set $\xi[v] = k + i$.

In particular, $\xi[u] = \xi[v]$ if and only if $(\xi[u_1], \ldots, \xi[u_d]) = (\xi[v_1], \ldots, \xi[v_d])$. Thus $\xi[u] = \xi[v]$ if and only if $T_u$ and $T_v$ are isomorphic.

### 2.3 On the Implementation

A straightforward implementation of the above procedure requires performing for each height $h$ a lexicographic sort of lists consisting of the children of vertices of height $h$. Let $n_h$ denote the number of vertices of height $h$. Then comparison-based methods require $O(n_{h-1} \log n_{h-1})$ time to sort all the lists of level $h$. Hence the total time will be $O(n \log n)$. Since the numbers to be sorted are in the range $1..n$, bucket-sort based methods can ensure only $O(n)$ time for each sort. In Figure 1(a) both the range of indexes and the height are $\theta(n)$; thus such an algorithm requires time $\Omega(n^2)$. In the next section we describe a linear time algorithm.

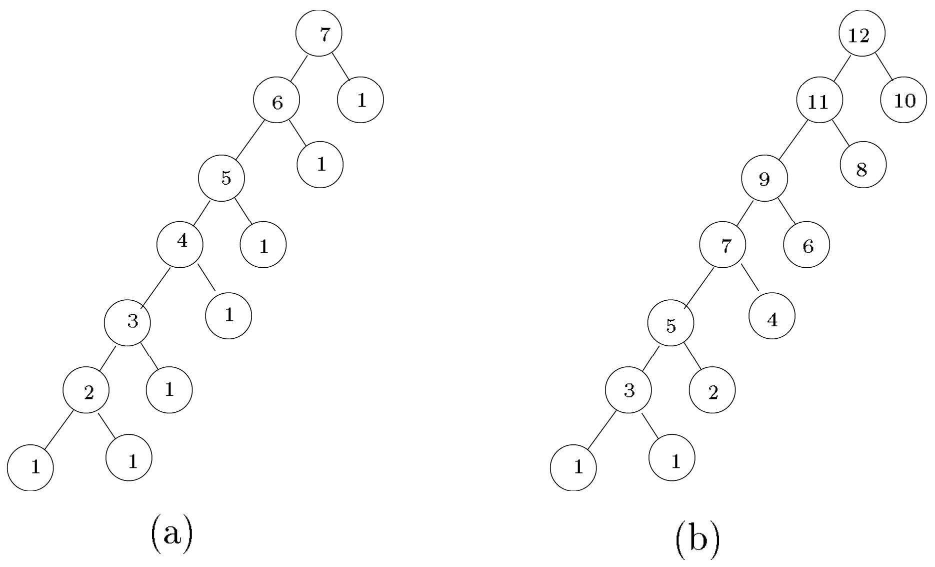

Figure 1: Bad tree for algorithm.

## 3 Canonical Indexes in Linear Time

### 3.1 Overview of the Algorithm

Let $level[h]$ consist of all the vertices of height $h$. We construct the canonical indexes (indexes, for short) level by level, starting with $level[0]$—the leaves. When processing $level[h]$ we assume that we have already found the indexes of all the vertices at lower levels and have constructed a single list, $\overrightarrow{children}[h]$, consisting of all the children of vertices at $level[h]$ sorted by their index (in nondecreasing order).

To assign the indexes to the vertices of $level[h]$ we construct an auxiliary ordered tree—$\mathbf{D}_h$, whose vertices are labeled by values of the indexes of the children of $level[h]$. (The data structures of the tree $\mathbf{D}_h$ will appear in bold font.) The children of any vertex $\mathbf{d} \in \mathbf{D}_h$ have distinct labels and are ordered by increasing label. Each vertex $v$ of $level[h]$ will be associated with a vertex $\mathbf{d} \in \mathbf{D}_h$ such that the path to $\mathbf{d}$ from **root**, the root of $\mathbf{D}_h$, is labeled with the indexes of $v$'s children. More specifically, if we order the children of $v$ by nondecreasing index, then the label of the $i$-th vertex on the path is equal to the index of the $i$-th child of $v$. (See Figure 2.)

Initially, $\mathbf{D}_h$ consists of a single vertex—**root**, to which all vertices of $level[h]$ are associated. In the course of the algorithm, new vertices are added to $\mathbf{D}_h$, and the vertices of level $h$ proliferate down the tree.

We now scan all the children of $level[h]$ by nondecreasing index. When considering a vertex $u$, we move its parent $v$ from $\mathbf{d}$, the $\mathbf{D}$-vertex to which it was associated, to $\mathbf{d}'$, the child of $\mathbf{d}$ whose label is equal to $\xi[u]$. If $\mathbf{d}$ has no such child, a new child is inserted. Since the children of $level[h]$ are processed by nondecreasing index, $\mathbf{d}'$ has the largest label of all the current children of $\mathbf{d}$. (This

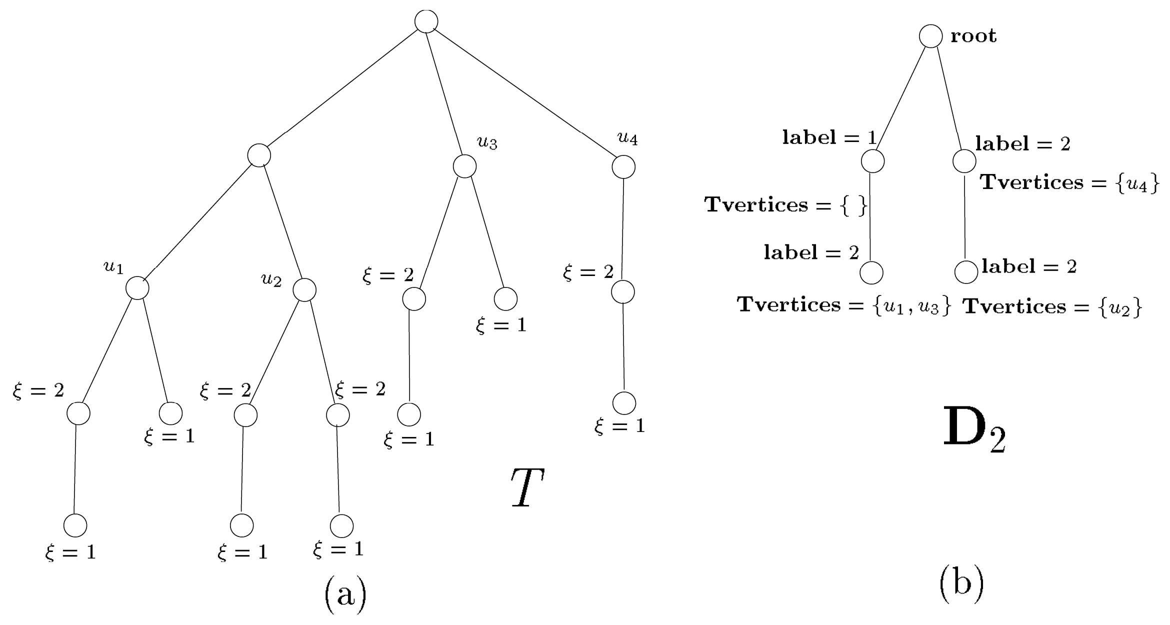

Figure 2: (a) The tree $T$, $level[2] = (u_1, u_2, u_3, u_4)$. (b) The tree $\mathbf{D}_2$.

observation eliminates the need to explicitly sort the children of vertex $\mathbf{d}$, thus allowing efficient insertions into $\mathbf{D}$.)

When completed, the structure of $\mathbf{D}_h$ reflects the canonical order of the vertices of $level[h]$: Let $u, v$ be vertices of $level[h]$ and let $\mathbf{d}_u$ ($\mathbf{d}_v$) be the $\mathbf{D}$-vertex associated with $u$ ($v$). $T_u$ is isomorphic to $T_v$ if and only if $\mathbf{d}_u = \mathbf{d}_v$. Also, $u$ precedes $v$ in the canonical order if and only if $\mathbf{d}_u$ precedes $\mathbf{d}_v$ in preorder. Therefore, we traverse $\mathbf{D}_h$ in preorder, assign the next consecutive integers as indexes to the vertices of $level[h]$ and construct the sorted list $\overrightarrow{level}[h]$ of these vertices.

Finally, we have to prepare the lists $\overrightarrow{children}[h+1], \ldots, \overrightarrow{children}[height(T)]$ of the children of vertices in levels $h+1, \ldots, height(T)$. In iteration $h$, we scan $\overrightarrow{level}[h]$: Let $v \in \overrightarrow{level}[h]$, $f$ be the parent of $v$ and $j > h$ be the level of $f$. We put $v$ at the end of the list of the children of $level[j]$. Note that while processing $level[h]$, we will have partial lists of children of levels $h+1, \ldots, height(T)$. The list of children of $level[j]$ will be completed only after processing $level[j-1]$. Since the vertices of $level[h]$ have greater indexes than those of $level[h']$ for any $h' < h$, and the vertices of $\overrightarrow{level}[h]$ are sorted by nondecreasing index, the lists of children are also sorted by nondecreasing index.

## 3.2 Detailed Description of the Algorithm

### Data structures:

1. For each vertex $v \in T$ we have a record with the following fields:

   $\xi$: the index of $v$.

   *parent*: the parent of $v$.

   *height*: the height of $v$.

   *Dvertex*: the vertex of $\mathbf{D}_h$ which currently contains $v$.

2. For each vertex $\mathbf{d} \in \mathbf{D}_h$ we have the following fields:

   **Tvertices:** the set of vertices of $T$ associated with $\mathbf{d}$.

   **rightmost:** the last child of $\mathbf{d}$.

   **label:** each vertex $v \in \mathbf{Tvertices}[\mathbf{d}]$ was moved from the parent of $\mathbf{d}$ to $\mathbf{d}$ when processing a vertex $c_v$, which is the $depth(\mathbf{d})$-th child of $v$ and has index $\xi[c_v]$; all such children have the same index, the value of which is recorded in **label**.

3. For $h = 0, \ldots, height(T)$ we have:

   $level[h]$**:** The set of the vertices of height $h$.

   $\overrightarrow{level}[h]$**:** The vertices of $level[h]$ sorted by nondecreasing index.

   $\overrightarrow{children}[h]$**:** The children of vertices of $level[h]$ sorted by nondecreasing index.

## Initialization

```
Scanning $T$ from the leaves upwards,
    determine for each $v \in V(T)$ the value of $height[v]$, and $parent[v]$;
For $h = 0, \ldots, height(T)\{$
    determine $level[h]$;
    Set $\overrightarrow{children}[h] = empty\_list;\}$
```

## The main loop

```
For $h = 0, \ldots, height[T]$ do
    Phase $h$
```

## Phase 0

```
For all $v \in level[0]$ /* the leaves of $T$ */
        set $\xi[v] = 1$ and append $v$ to $\overrightarrow{children}[height[parent[v]]]$.
set $max\xi = 1$
```

## Phase h ($h \geq 1$)

Assumptions: We have already constructed $\overrightarrow{children}[h]$.

```
M1:  Construct $\mathbf{D}_h$ from $\overrightarrow{children}[h]$. (See below.)
M2:  Traverse $\mathbf{D}_h$ to determine $\xi[v]$, $v \in level[h]$,
         and construct $\overrightarrow{level}[h]$. (See below.)
M3:  for $v \in \overrightarrow{level}[h]$ do   /* $\overrightarrow{level}[h]$ should be processed by increasing order */
         append $v$ to the list $\overrightarrow{children}[height[parent[v]]]$.
```

**Construct $\mathbf{D}_h$ from $\overrightarrow{children}[h]$.**

```
C1:     Initially $\mathbf{D}_h$ consists of a single vertex—$\mathbf{root}$.
C2:     Let $\mathbf{Tvertices}[\mathbf{root}] = level[h]$;
C3:     for $v \in level[h]$ do
C3.1        $Dvertex[v] = \mathbf{root}$;
C4:     for $u \in \overrightarrow{children}[h]$ do   /* $\overrightarrow{children}[h]$ should be processed by increasing order */
C4.1:       let $v = parent[u]$;
C4.2:       let $\mathbf{d} = Dvertex[v]$;
C4.3:       if $\mathbf{d}$ is a leaf or $\mathbf{label}[\mathbf{rightmost}[\mathbf{d}]] \neq \xi[u]$ then
C4.3.1:         add a new child $\mathbf{d}'$ to $\mathbf{d}$ ;
C4.3.2:         $\mathbf{label}[\mathbf{d}'] = \xi[u]$;
C4.3.3:         $\mathbf{rightmost}[\mathbf{d}] = \mathbf{d}'$;
C4.4:       else /* $label[rightmost[d]] = \xi[u]$ */
                $\mathbf{d}' = \mathbf{rightmost}[\mathbf{d}]$;
C4.5:       move $v$ from $\mathbf{d}$ to $\mathbf{d}'$ /* update $Dvertex[v]$, $\mathbf{Tvertices}[\mathbf{d}]$ and $\mathbf{Tvertices}[\mathbf{d}']$ */.
```

**Traverse $\mathbf{D}_h$:**

```
T1: Set $\overrightarrow{level}[h] = empty\_list$;
T2: traverse $\mathbf{D}_h$ in pre-order;
T3: on arrival at a vertex $\mathbf{d} \in \mathbf{D}_h$ such that $\mathbf{Tvertices}[\mathbf{d}] \neq \emptyset$ do
T3.1:     $max\xi = max\xi + 1$;
T3.2:     for all $v \in \mathbf{Tvertices}[\mathbf{d}]$ do
T3.2.1:       $\xi[v] = max\xi$;
T3.2.2:       append $v$ to $\overrightarrow{level}[h]$.
```

## 4 Hopcroft and Tarjan's Canonical Order

The canonical order of Section 2.1 is defined from the leaves up. The same holds for the canonical indexes. As a result, two vertices $u$ and $v$ have the same canonical index if and only if $T_u$ and $T_v$ are isomorphic.

Hopcroft and Tarjan's work [4] is aimed at tree isomorphism rather than subtree isomorphism. In their construction, the *depth*—the distance from the root—also distinguishes between vertices. Consequently, they assign the same index to vertices $u$ and $v$ if and only if they have the same depth *and* $T_u$ and $T_v$ are isomorphic (see Figure 1(b) for an example of their index).

This choice enables Hopcroft and Tarjan to overcome the difficulty (as mentioned in Section 2.3) of using bucket sort. They process the tree vertices by phases: in phase $i$ they process all vertices of depth $d_{max} + 1 - i$. Since indexes are given in increasing order, the indexes of all the vertices of depth $d$ form a set of consecutive integers. Notice that all children of vertices at the depth $d$ are at the depth $d+1$. Thus, the sets of children in distinct phases are disjoint. Therefore, the sum of their sizes is at most $n$, not $\Theta(n^2)$, as in the naive height oriented approach of Section 2.3.

Hopcroft and Tarjan's algorithm yields a refinement of the subtree isomorphism classes, as generated by Zemlyachenko's canonical indexes. Given the subtree isomorphism classes, it is easy to construct Hopcroft and Tarjan's classes. Other than applying our algorithm, we do not know how to perform the reverse direction.

## 5 An Application

One possible application of the approach of this paper is as follows. Tinhofer and Schreck [7] suggested a code (compact representation) for trees, which can be computed in linear time and which they claim to be the most compact among known codes. They used the indexing approach of Hopcroft and Tarjan. In the case that there are several isomorphic subtrees hanging on vertices at the same distance from the root, they delete from the code descriptions of all such subtrees, except for one, retaining only the index of the vertex on which the deleted subtree hanged. However, subtree isomorphic vertices of different depths are not identified by their procedure.

Using Zemlyachenko's isomorphism procedure, which disregards distance from the root, instead of Hopcroft and Tarjan's, would allow to identify all subtree isomorphic vertices. Thus the number of deleted subtrees could increase resulting in a more compact code. This approach is generalize and further investigated in a companion paper [3].

## References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] R. G. Busacker and T. L. Saaty. *Finite graphs and networks.* McGraw-Hill, New York, 1965.

[3] Y. Dinitz, A. Itai, and M. Rodeh. Parsing of trees. (In preparation).

[4] J. E. Hopcroft and R. E. Tarjan. Isomorphism of planar graphs. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 131–150. Plenum Press, 1972.

[5] J. Lederberg. Dendral-64: A system for computer construction, enumeration and notation of organic molecules as tree structures and cyclic graphs, Part I: Notation algorithm for tree structures. Interim Report NASA Cr 68898, STAR No. N-66-14074, National Aeronautics and Space Administration, 1964.

[6] H. I. Scions. Placing trees in lexicographic order. *Machine Intelligence*, 3:43–60, 1968.

[7] G. Tinhofer and H. Schreck. Linear time tree codes. *Computitng*, 33:211–225, 1984.

[8] V. N. Zemlyachenko. Determining tree isomorphism. In *Voprosy Kibernetiki, Proceedings of the Seminar on Combinatorial Mathematics (Moscow 1971)*, pages 54–60. Scientific Council on the Complex Problem "Kibernetika", Akad. Nauk SSSR, 1973. (In Russian).