Computing and Informatics, Vol. 33, 2014, 1428–1450

GENERATION OF NEURONAL TREES BY A NEW THREE LETTERS ENCODING

Mahdi Amani, Abbas Nowzari-Dalini, Hayedeh Ahrabian

Department of Computer Science
School of Mathematics, Statistics, and Computer Science
University of Tehran, Tehran, Iran
&
Institute for Studies in Theoretical Physics and Mathematics (I.P.M.)
Tehran, Iran
e-mail: {`mahdi.amani, nowzari, ahrabian`}`@ut.ac.ir`

Abstract. A neuronal tree is a rooted tree with n leaves whose each internal node has at least two children; this class not only is defined based on the structure of dendrites in neurons, but also refers to phylogenetic trees or evolutionary trees. More precisely, neuronal trees are rooted-multistate phylogenetic trees whose size is defined as the number of leaves. In this paper, a new encoding over an alphabet of size 3 (minimal cardinality) is introduced for representing the neuronal trees with a given number of leaves. This encoding is used for generating neuronal trees with n leaves in A-order with constant average time and $O(n)$ time complexity in the worst case. Also, new ranking and unranking algorithms are presented in time complexity of $O(n)$ and $O(n \log n)$, respectively.

Keywords: Neuronal tree, evolutionary tree, phylogenetic tree, tree of life, cladistic rooted tree, generation algorithm, ranking and unranking algorithms

Mathematics Subject Classification 2010: 05C05, 05C85, 68R10

1 INTRODUCTION

Many papers have been published earlier in the literature for generating different classes of trees. For example, we can mention the generation of binary trees in [1, 2, 3, 4, 5, 6, 7], k-ary trees in [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], trees with n nodes and m leaves in [19, 20], neuronal trees in [21, 22], AVL trees in [23], and search tree in [24].

Typically, trees are encoded as strings over a given alphabet and then these strings (called codeword) are generated. By choosing a suitable codeword to represent the trees, we can design efficient generation algorithm for these codewords. Any generation algorithm imposes an ordering on the set of trees. The best known orderings on trees are A-order and B-order [25]. The A-order definition uses global information concerning the tree nodes and appear to be a natural ordering of trees, whereas the B-order definition uses local information .

Neuronal trees are a type of trees which have n leaves and each internal node has at least two children. These trees are known with regard to their number of leaves [21]. The neuronal tree can be considered as a data structure for modeling the "dendrites of a nerve cell" [21] and "phylogenetic trees" or "evolutionary trees" which is based on branching diagram which shows the evolutionary relationships among various biological species (species are the leaves and internal nodes are hypothetical ancestors, as they cannot be directly observed) [26, 27, 28].

Generation of this type of trees is first studied by Pallo [21]. He introduced an integer sequence codeword for these trees and presented a generation algorithm for neuronal trees with n leaves in B-order with $O(n)$ worst case time complexity without ranking and unranking algorithms. An encoding of length n over six letters for a neuronal tree with n leaves and a generation algorithm on this encoding in A-order are given by Vajnovszki [22]. The presented generation algorithm has $O(n)$ time complexity in the worst case and $O(\log n)$ average time complexity. He also presented an unranking algorithm with the time complexity of $O(n^2)$ but no ranking algorithm was presented. It seems to be quite challenging to decrease the average time complexity of generation algorithm and design a ranking algorithm for them and also improve the time complexity of the unranking algorithm.

In this paper, we present a new encoding on three letters for the neuronal trees with n leaves. The size of our encoding is equal to the number of nodes of the tree (less than $2n$ and greater than $n + 1$). We also present a generation algorithm on this encoding with constant average time, $O(1)$, and $O(n)$ time complexity in the worst case. In this algorithm, the trees are generated in A-order. Due to the given encoding, both ranking and unranking algorithms are also presented with $O(n)$ and $O(n \log n)$ time complexity, respectively.

2 DEFINITION

Here we assume that the reader is familiar to the definitions of graph, tree, node, internal node, leaf, size, depth, child and subtree in trees, and generation, ranking and unranking algorithms. However here we define some important concepts in neuronal trees.

According to the structure of neurons, dendrites of neurons have two or more splits in the way that each branch in the splits is connected to at least two other branches. Therefore, dendrite trees that are also called neuronal trees are defined as trees in which each internal node has at least two children; in other words a neuronal tree is a tree whose each node is either a leaf or has at least 2 children [21, 22].

Formally, a neuronal tree T is defined as a finite set of one or more nodes such that:

1. T has a distinguished node r, called root of this tree, and if T has more than one node, then r is connected to $j \geq 2$ neuronal trees $T_1, T_2, \ldots, T_j$ and each tree T_i $(1 \leq i \leq j)$, is called the subtree of T.
2. The root of T_i $(1 \leq i \leq j)$ is considered as a child of r.
3. T_1 is the leftmost subtree, and its root is the leftmost child of r.
4. T_j is the rightmost subtree and its root is the rightmost child of r.

Recall from [21, 22]: in a tree, the degree of a node is equal to the number of its children, and the degree of the tree is equal to maximum degree of its nodes. An example of a neuronal tree with 9 leaves is shown in Figure 1.

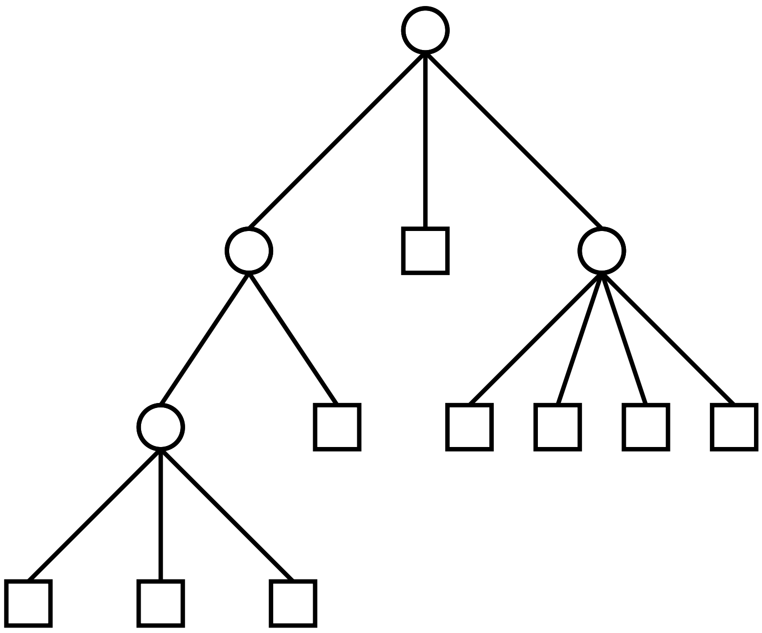


Figure 1. An example of a neuronal tree with 9 leaves

Let S_n be the cardinality of set of neuronal trees with n leaves; recalling from [22] the following relations are presented:

$$\begin{aligned} S_1 &= S_2 = 1, \\ nS_n &= 3(2n-3)S_{n-1} - (n-3)S_{n-2}, \text{ for all } n \geq 3. \end{aligned}$$

It is also proved in [22] that $5^n < S_n$ for $n > 57$ and $S_n < 6^n$ for $n > 1$.

The number of neuronal trees is equivalent to the number of some important combinatorial objects which are introduced below:

1. The number of "super-Catalan numbers" that is also called "generalized parentheses" or "little Schröder numbers". This number is the number of ways to insert parentheses in a string of $n + 1$ symbols such that the parentheses must be balanced but there is no restriction on the number of pairs of parentheses, except that the number of letters or sub-parentheses inside a pair of parentheses must be at least 2. Obviously parentheses enclosing the whole string are ignored [29, 30, 31, 32].
2. The number of Schröder paths of semilength $n - 1$ (i.e. lattice paths from $(0, 0)$ to $(2n - 2, 0)$, with steps $H = (+2, 0)$, $U = (+1, +1)$ and $D(+1, -1)$ and not going below the x-axis) with no peaks at level 1 [29].

Up to now, no generation algorithms have been presented for these combinatorial objects and obviously the generation of neuronal trees is equivalent to the generation of these combinatorial objects.

As mentioned, any generation algorithm imposes an ordering on the set of trees. Two such natural orderings are A-order and B-order [3, 22, 25] which are defined as follows.

Definition 1. Let T and T' be two neuronal trees and $k = max\{\deg(T), \deg(T')\}$; we say that T is less than T' in B-order ($T \prec_B T'$), iff

- $\deg(T) < \deg(T')$, or
- $\deg(T) = \deg(T')$ and for some $1 \leq i \leq k$, $T_j =_B T'_j$ for all $j = 1, 2, \ldots, i-1$, and $T_i \prec_B T'_i$

where $\deg(T)$ is defined as the degree of root of the tree T.

Definition 2. Let T and T' be two neuronal trees and $k = \max\{\deg(T), \deg(T')\}$; we say that T is less than T' in A-order ($T \prec_A T'$), iff

- $|T| < |T'|$, or
- $|T| = |T'|$ and for some $1 \leq i \leq k$, $T_j =_A T'_j$ for all $j = 1, 2, \ldots, i-1$ and $T_i \prec_A T'_i$

where $|T|$ (size of T) is defined as the number of leaves in the tree T.

As mentioned earlier, in most generation algorithms trees are encoded as strings over a given alphabet and then these sequences are generated in specified order such that their corresponding trees are in A-order or B-order. Our generation algorithm, which is given in the next section, produces the sequences such that their corresponding trees are in A-order.

3 THE ENCODING SCHEMA AND GENERATION ALGORITHM

The main point in generating trees is to choose a suitable encoding to represent them, and instead of generating trees, their corresponding codewords are generated. In this section, we give an encoding for neuronal trees on 3 letters and later we present an algorithm that generates the successor sequence of a given codeword of a neuronal tree with n leaves in A-order.

Based on what we mentioned earlier, S_n is the cardinality of set of neuronal trees with n leaves and we know $S_n < 6^n$. Clearly $S_n < 3^{2n}$ and the number of nodes of a neuronal tree with n leaves is less than $2n$ (it can be shown that total number of nodes in a neuronal tree with n leaves is at least $n+1$ and at most $2n-1$), therefore it is desirable to encode these trees with 3 letters. Now, we prove that if the size of codeword be less than the number of neuronal tree nodes, then the encoding of neuronal trees with two letters is impossible.

Lemma 1. The minimum number of letters required for encoding the set of neuronal trees with n leaves is 3, if the size of codeword be less than the number of neuronal tree nodes.

Proof. It can be shown that the total number of nodes in a neuronal tree with n leaves is at least $n+1$ and at most $2n-1$, so the number of codewords over two letters is at most equal to:

$$2^{n+1} + 2^{n+2} + \ldots + 2^{2n-1} = 2^{n+1}(2^{n-1} - 1) = 2^{2n} - 2^{n+1} < (2+2+1)^n < 5^n < S_n.$$

Consequently, we can deduce that the least size of alphabet for encoding neuronal trees is 3. Obviously, a neuronal tree codeword over 2 letters has a length greater than $2n$. □

Regarding the above properties, we present our new encoding. For any neuronal tree T with n leaves, the encoding over three letters $\{\ell, m, r\}$ is defined as follows. The root of T is labeled by 'm', the leftmost child of any internal node is labeled by 'ℓ', the rightmost child of any internal node is labeled by 'r', and the children between leftmost and rightmost children (if exist) are labeled by 'm'. This labeling is proceeded in the same way for any internal node in each level recursively and a preorder traversal of T will result in the codeword. This labeling is illustrated in Figure 2 for a given tree T, and the codeword corresponding to this tree is "$m\ell\ell\ell mrr\ell rmm\ell mmrr$". Note that the 3-letter alphabet codeword corresponding

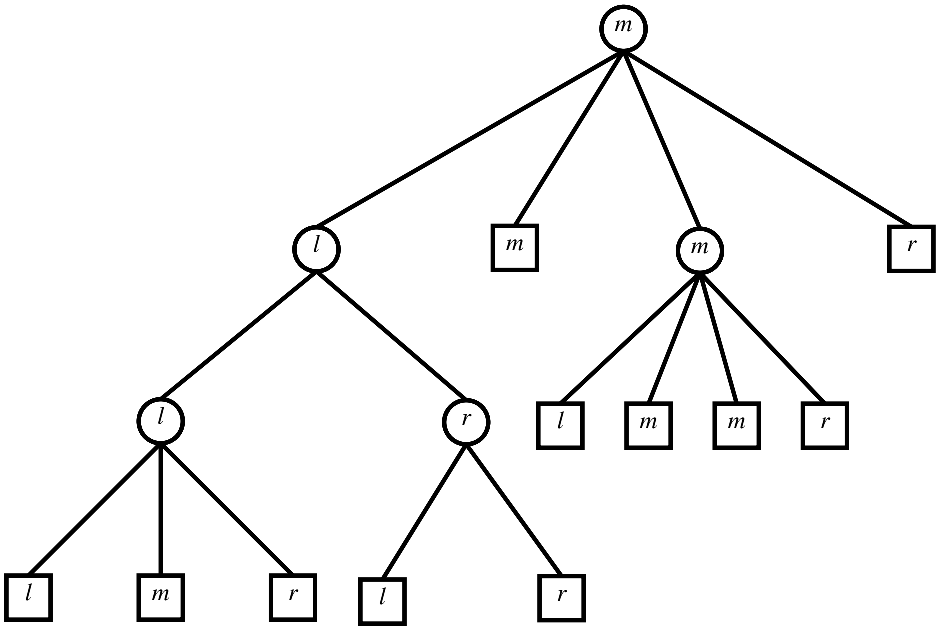


Figure 2. An example of labeled neuronal tree

to the first and last neuronal tree with n leaves in A-order is respectively "$m\ell m^{n-2}r$" and "$m\ell^{n-1}r^{n-1}$".

Now we prove the validity of a given string on three letters as a neuronal tree encoding (the one-to-one correspondence between the encoding and the tree). A valid codeword is defined by the following definition and theorem.

Definition 3. Suppose that $\{\ell, m, r\}^*$ is the set of all sequences with alphabet of ℓ, m, r and let A be a proper subset of $\{\ell, m, r\}^*$; then we call the set A a "CodeSet" iff A satisfies the following properties:

1. $\epsilon \in A$ (ϵ is a string of length 0),
2. $\forall x_1, x_2, \ldots, x_i \in A$, and $i \geq 2$: $\ell x_1 m x_2 m x_3 \ldots m x_{i-1} r x_i \in A$.

Theorem 1. Let A be a "CodeSet", δ be a string such that $\delta \in A$ and C be a codeword obtained by the concatenation of the character $'m'$ and δ (we show it by $m\delta$). There is a one-to-one correspondence between C and a unique neuronal tree.

Proof. It can be proved by induction on the length of C. Initially for a codeword of length equal to 1 the proof is trivial. Assume each codeword obtained in the above manner with length less than n encodes a unique neuronal tree. For any given codeword with length n we have:

$$C = m\ell x_1 m x_2 \ldots m x_{j-1} r x_j, \text{ such that } x_i \in A, \forall 1 \leq i \leq j.$$

By induction hypothesis, each mx_i for $1 \leq i \leq j$ is a valid codeword for a neuronal tree; therefore with replacement of m with ℓ in mx_1 and also m with r in mx_j we can produce the ℓx_1 and rx_j codewords. Thus they all are subtrees of a neuronal tree whose codeword is $C = m\ell x_1 mx_2 \ldots mx_{j-1} rx_j$. This tree is shown in Figure 3.

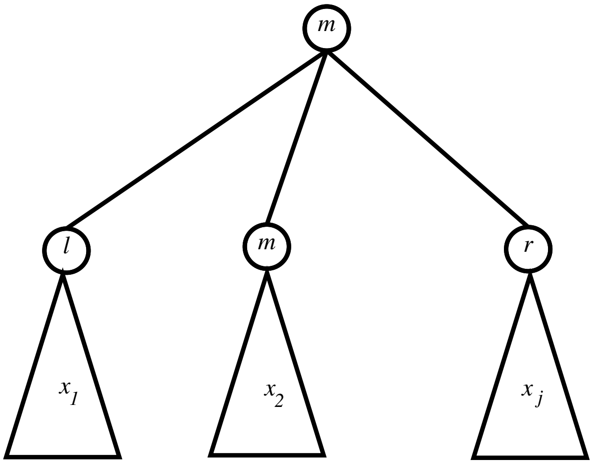


Figure 3. The neuronal tree encoded by $C = m\ell x_1 mx_2 \ldots mx_{j-1} rx_j$

Reversely, we assume each neuronal tree T with size $k < n$ has a unique code in the form $C = m\ell x_1 mx_2 \ldots mx_{j-1} rx_j$. Now, we consider the tree T with size n as shown in Figure 3. Employing the induction assumption, the codeword corresponding to the first subtree of this tree (Figure 3) is equal to ℓx_1 where $x_1 \in A$ and the codeword corresponding to the second subtree is equal to mx_2 where $x_2 \in A$, $\ldots$, and the codeword corresponding to the j^{th} subtree is equal to rx_j where $x_j \in A$. With respect to the definition of A, the concatenation of these codewords ($\delta = \ell x_1 mx_2 \ldots mx_{j-1} rx_j$) belongs to A. Clearly $C = m\delta$ represents a codeword of a tree composed of the above subtrees. □

Now, for generating the successor of a given codeword C corresponding to neuronal tree T, the codeword C is scanned from right to left. Scanning the codeword C from right to left, corresponds to reverse preorder traversal of labeled neuronal tree T. During the scan, either one of the characters $'m'$, $'\ell'$ or $'r'$ is visited, and on the corresponding node to the visited character in tree T the following operations are proceeded.

1. Visiting character 'r' means we have arrived to a rightmost node. This node can not be extended individually. By passing this node we continue the scanning.
2. Visiting character 'm' means that we have arrived to a subtree in between. Therefore this subtree can be extended. Definitely there exists a co-level leaf which is previously scanned and can be added to the subtree of current node (node corresponding to the visited character) and the scanning is terminated.

3. Visiting character 'ℓ' means that we have arrived to the leftmost subtree. In this case, we know the right brother of the current node can not be labeled by 'm', because scanning each visited character 'm', means that the successor had been produced, so definitely its right brother must be labeled by 'r'. Therefore it can not be added to the subtree of the current node because the number of their siblings will be one, and it is against the definition of neuronal tree. Now we must connect both of them directly to their ascendants.

The pseudo code for this algorithm is presented in Algorithm 1.

In this algorithm, array C is a global array of characters holding the codeword (the algorithm generates the successor sequence of this codeword), $Size$ shows the size of codeword and N is a local array of size $|C|$ where $N[i]$ shows the size of subtree rooted by the node corresponding to $C[i]$. This array is computed implicitly during the performance of the algorithm. The variable b shows the index of right sibling and the variable w is a local variable for saving the value of $N[i]$ during the performance of the algorithm.

In order to generate all neuronal trees of size n, we start from "$m\ell m^{n-2}r$" which is the codeword corresponding to the first neuronal tree in A-order and we recall the algorithm for S_n times.

Theorem 2. The algorithm Next presented in Algorithm 1 has a worst case time complexity of $O(n)$ and an average time complexity of $O(1)$.

Proof. Worst case time complexity of this algorithm is $O(n)$ because the sequence is scanned just once. For computing average time, it should be noted that during the scanning process every time we visit the character 'm', the algorithm will be terminated, so we define S_i^n as the number of codewords of neuronal trees with n leaves whose last character 'm' has distance i from the end. Obviously we have:

$$S_n = \sum_{i=1}^{2n-1} S_i^n.$$

Now, consider that for S_i^{n+1} we have two cases, the last character 'm' may be a leaf or not. So obviously S_i^{n+1} is greater than just the first case and by removing the character 'm' we have:

$$S_i^{n+1} \geq \sum_{j=i}^{2n-1} S_j^n.$$

We define H_n as the average time of generating all codewords of neuronal trees with n leaves, so:

$$\begin{aligned} H_n &\leq (C/S_n) \sum_{i=1}^{2n-1} iS_i^n, \\ H_n &\leq (C/S_n) \sum_{j=1}^{2n-1} \sum_{i=j}^{2n-1} S_i^n, \end{aligned}$$

Algorithm 1 The Next algorithm

```
Function Next (Size: integer)
var $w$, $i$, $b$, $k$, $j$: integer;
begin
    $i := Size$; $N[i] := 1$;
    repeat
        $b := i$; $i := i - 1$;
        $N[i] := 1$;
        while (($C[i] = \ell$) and ($C[b] = r$) and ($N[b] = 1$)) do begin
            $i := i - 1$; $N[i] := N[i + 1] + 1$;
            $N[i + 1] := 0$; $N[b] := 0$; $b := b + 1$;
        end;
    until ($C[i] = m$) or ($i = 1$) or (($C[i] = \ell$) & ($C[b] = r$) & ($N[b] > 1$));
    if ($i = 1$) then
        return(false)
    else
        if (($C[i] = m$) and ($N[b] = 1$)) then $C[i] := C[b]$;
    $w := N[i]$; $N[i] := 0$; $i := i + 1$; $C[i] := \ell$;
    for $k := 1$ to $w - 1$ do begin
        $i := i + 1$; $C[i] := m$;
    end;
    $i := i + 1$; $C[i] := r$;
    if $N[b] > 1$ then begin
        for $k := 1$ to ($N[b] - 2$) do begin
            $i := i + 1$; $C[i] := m$;
        end;
        $N[b] := 0$;
        $i := i + 1$; $C[i] := r$;
    end;
    for $j := b + 1$ to $Size$ do begin
        if $N[j] > 0$ then begin
            for $k := 1$ to $N[j] - 1$ do begin
                $i := i + 1$; $C[i] := m$;
            end;
            $N[j] := 0$;
            $i := i + 1$; $C[i] := r$;
        end;
    end;
    $Size := i$;
    return(true);
end;
```

$$\begin{aligned} H_n &\leq (C/S_n) \sum_{j=1}^{2n-1} S_j^{n+1}, \\ H_n &\leq CS_{n+1}/S_n, \\ H_n &\leq C \times 6 = O(1) \end{aligned}$$

where C is a constant value.

Finally, it should be mentioned that this constant average time complexity is obviously without considering the consumed time for input and output. □

4 RANKING AND UNRANKING ALGORITHMS

By designing a generation algorithm in a specific order, the rank of a tree is its position in the exhaustive generated list. Rank of a neuronal tree with respect to some ordering is the number of neuronal trees that come before it in the ordering. Unranking is as usual the inverse of the ranking. An unranking algorithm determines the neuronal tree having a particular rank. In this section, ranking and unranking algorithms for neuronal trees in A-order are given.

The following theorems and definitions help us in designing the rank algorithm in A-order.

Theorem 3. Let S_n be the cardinality of neuronal trees with n leaves, then

$$\begin{aligned} S_n &= 2\sum_{i=1}^{n-2} S_i S_{n-i} + S_{n-1}S_1, \\ S_1 &= S_2 = 1. \end{aligned}$$

Proof. Let T be a neuronal tree with n leaves; we have two different cases (remember that each internal node has at least two children):

1. T has just two children T_1 with i leaves and T_2 with $n-i$ leaves. Obviously in this case the total number of neuronal trees can be enumerated by the following formula:
$$\sum_{i=1}^{n-1} S_i S_{n-i}.$$
2. T has $k > 2$ children $T_1, T_2, \ldots, T_k$ such that T_1 has i leaves and T_2, T_3, …, T_k have all together $n-i$ leaves. Since $k > 2$, if we ignore T_1, a neuronal tree with $n-i$ leaves remains, and because the value of i can not be equal to $n-1$, so the total number of trees in this condition is:
$$\sum_{i=1}^{n-2} S_i S_{n-i}.$$

Now by summing up these cases we have:

$$S_n = 2\sum_{i=1}^{n-2} S_i S_{n-i} + S_{n-1}S_1.$$

$\square$

Theorem 4. Let $D_{n,i}$ be the number of neuronal trees with n leaves whose first subtree has at least 1 and at most i leaves. Then we have:

1. If $i < n-1$ then $D_{n,i} = 2\sum_{j=1}^{i} S_j S_{n-j}$.
2. If $i = n-1$ then $D_{n,i} = 2\sum_{j=1}^{n-2} S_j S_{n-j} + S_{n-1}$.

Proof. Let T be a neuronal tree with n leaves; we have two different cases:

1. If $i < n-1$ then we have two cases again:

   a) T has just two children T_1 and T_2. Obviously in this case the total number of neuronal trees whose first subtree has at least 1 and at most i leaves can be enumerated by the following formula:

   $$\sum_{j=1}^{i} S_j S_{n-j}.$$

   b) T has $k > 2$ children $T_1, T_2, \ldots, T_k$ such that T_1 has at least 1 and at most i leaves and T_2, T_3, …, T_k have totally the remaining leaves. Since $k > 2$, if we ignore T_1, what remains is a neuronal tree too. So the total number of trees whose first subtree has at least 1 and at most i leaves in this case is:

   $$\sum_{j=1}^{i} S_j S_{n-j}.$$

   Now by summing up these cases we have:

   $$D_{n,i} = 2\sum_{j=1}^{i} S_j S_{n-j}.$$

2. If $i = n-1$ then we have two cases again:

   a) T has just two children T_1 and T_2. Obviously in this case the total number of neuronal trees whose first subtree has at least 1 and at most i leaves can be enumerated by the following formula:

   $$\sum_{j=1}^{n-1} S_j S_{n-j}.$$

b) T has $k > 2$ children $T_1, T_2, \dots, T_k$ such that T_1 has at least 1 and at most i leaves and T_2, T_3, …, T_k have the remaining leaves. Since $k > 2$, it is impossible that the first subtree has $n-1$ leaves because T_2, T_3, …, T_k totally have at least 2 leaves (each one has at least one leaf). In this case again if we ignore T_1, what remains is a neuronal tree too. So the total number of trees whose first subtree has at least 1 and at most i leaves in this case is:

$$\sum_{j=1}^{n-2} S_j S_{n-j}.$$

Now by summing up these cases we have:

$$D_{n,i} = 2\sum_{j=1}^{n-2} S_j S_{n-j} + S_{n-1}.$$

Hence, the proof is complete. □

For ranking and unranking algorithms we need S_n and $D_{n,i}$ that are defined and computed earlier. We assume these values are computed and stored in arrays $S[n]$ and $D[n,i]$. The pseudo code given in Algorithm 2 constructs and stores them in arrays $S[n]$ and $D[n,i]$ in $O(n)$ and $O(n^2)$.

Algorithm 2 Algorithm for producing $S[n]$ and $D[n,i]$ arrays

```
Function produceSD ($n$: integer)
var $i$, $j$, $k$: integer;
begin
    $S[0] := 1$; $S[1] := 1$;
    for $i := 2$ to $n$ do begin
        for $j := 1$ to $(i-2)$ do
            $S[i] := S[i] + 2 \times S[j] \times S[i-j]$;
        $S[i] := S[i] + S[i-1]$;
    end;
    for $i := 1$ to $n$ do begin
        $D[i,0] := 0$;
        for $j := 1$ to $(i-2)$ do
            for $k := 1$ to $j$ do
                $D[i,j] := D[i,j] + 2 \times S[k] \times S[i-k]$;
        for $k := 1$ to $i-2$ do
            $D[i,i-1] := D[i,i-1] + 2 \times S[k] \times S[i-k]$;
        $D[i,i-1] := D[i,i-1] + S[i-1]$;
    end;
    $S[0] = 1/2$; $S[1] = 1/2$;
end;
```

Note that in Algorithm 2 initially we assign $1/2$ to the $S[0]$ and $S[1]$ instead of 1 because in the following ranking formula the values of coefficient of $S[0]$ and $S[1]$ are half of the other values; also remember that $S[n]$ and $D[n,i]$ arrays are computed just one time before running the main "Rank" or "Unrank" algorithms.

Let T be a neuronal tree with n leaves whose subtrees are defined by T_1, T_2, ..., T_j and for $1 \le i \le j : |T_i| = n_i$, and $\sum_{i=1}^{j} n_i = n$. For computing the rank of T, we have to enumerate the number of trees generated before T.

The number of neuronal trees with n leaves whose first subtree is smaller than T_1 is equal to:

$$D[n, n_1 - 1] + 2(\mathit{Rank}(T_1) - 1)S[n - n_1],$$

and the number of neuronal trees with n leaves whose first subtree is equal to T_1 but the second subtree is smaller than T_2 is equal to:

$$D[n - n_1, n_2 - 1] + 2(\mathit{Rank}(T_2) - 1)S[n - n_1 - n_2].$$

Similarly the number of neuronal trees with n leaves whose first $(j-1)$ subtrees are equal to $T_1, T_2, \ldots, T_{j-1}$ and the j^{th} subtree is smaller than T_j is equal to:

$$D[n - \sum_{k=1}^{j-1} n_k, n_j - 1] + 2(\mathit{Rank}(T_j, n_j) - 1).$$

Therefore, regarding the above enumeration we can write:

$$\begin{aligned} \mathit{Rank}(T,1) &= 1 \\ \mathit{Rank}(T,n) &= 1 + \sum_{i=1}^{j} \left(D\left[n - \sum_{k=1}^{i-1} n_k, \sum_{k=1}^{i} n_k - 1 \right] \right. \\ &\qquad \left. + 2(\mathit{Rank}(T_i, n_i) - 1) S\left[\sum_{k=1}^{i} n_k \right] \right). \end{aligned}$$

For the rank of a codeword stored in array C, we need an auxiliary array $N[i]$ which keeps the number of leaves in the subtree whose root is labeled by $C[i]$ and corresponds to the n_i in the above formula. This array is computed by the algorithm given in Algorithm 3. In this algorithm "*Beg*" is a variable that shows the positions of the first character in the array C, and "*Fin*" shows the position of last leave in subtree whose root is labeled by $C[\mathit{Beg}]$. This algorithm is recursive and in each call, for a codeword in global array C, the number of leaves of a subtree rooted at $C[\mathit{Beg}]$ with the last leaf in $C[\mathit{Fin}]$ is calculated. This algorithm is performed just once before calling the ranking algorithm.

The ranking algorithm is given in Algorithm 4.

In this algorithm, C is a global array which stores the codeword, N is global array which stores the number of leaves (calculated by Algorithm 3), *Beg* is also the variable that shows the positions of the first character in the array C whose rank is

Algorithm 3 Algorithm for calculating the number of leaves in a subtree

```
Function CalculateN($Beg$: integer)
var $Fin$, $Sum$, $Cur$: integer;
begin
    if $C[Beg+1] \neq \ell$ then begin
        $N[Beg] := 1$;
        return($Beg+1$);
    end else
    begin
        $Fin := Beg+1$; $Sum := 0$;
        repeat
            $Cur := Fin$;
            $Fin :=$ CalculateN($Cur$); $Sum := Sum + N[Cur]$;
        until ($C[Cur] = r$);
        $N[Beg] := Sum$;
        return($Fin$);
    end;
end;
```

being computed (*Beg* is initially set to 1), and *Fin* is the variable that returns the position of the last characters of C.

Now the time complexity of this algorithm is discussed. Obviously the time complexity of procedure "CalculateN" (presented in Algorithm 3) which computes the number of leaves in each subtree is $O(n)$. Since this algorithm is performed just once before calling the ranking algorithm and it has no more time effects on ranking algorithm, therefore we should calculate the time complexity of ranking algorithm.

Theorem 5. The ranking algorithm has the time complexity of $O(n)$.

Proof. Let T be a neuronal tree with n leaves whose subtrees are defined by T_1, T_2, …, T_j and for $1 \leq i \leq j : |T_i| = n_i$ and $\sum_{i=1}^{j} n_i = n$, and let $T(n)$ be the time complexity of ranking algorithm, then we can write:

$$T(n) = T(n_1) + T(n_2) + \ldots + T(n_j) + \alpha j,$$

where α is a constant and αj is the time complexity of the non-recursive parts of the algorithm.

By using induction, we prove if β be a value greater than α then $T(n) \leq \beta n$. We have $T(1) < \beta$. We assume $T(k) \leq \beta(k-1)$ for each $k < n$, therefore

$$\begin{aligned}
T(n) &\leq \beta(n_1 - 1) + \beta(n_2 - 1) + \ldots + \beta(n_j - 1) + \alpha j,\\
T(n) &\leq \beta(n_1 + \ldots + n_j - j) + \alpha j,\\
T(n) &\leq \beta n - \beta j + \alpha j,\\
T(n) &\leq \beta n.
\end{aligned}$$

So the induction is complete and we have:

$$T(n) \leq \beta n \Longrightarrow T(n) = O(n).$$

$\square$

Before giving the description of the unranking algorithm we need to define two new operators.

- If a and b are integer numbers then $a\, div^{+}\, b$ is defined as follows:
  - If $b \nmid a$ then $a\, div^{+}\, b$ is equal to $(a\, div\, b)$.
  - If $b \mid a$ then $a\, div^{+}\, b$ is equal to $(a\, div\, b) - 1$.
- If a and b are integer numbers then $a\, mod^{+}\, b$ is defined as follows:
  - If $b \nmid a$ then $a\, mod^{+}\, b$ is equal to $(a\, mod\, b)$.
  - If $b \mid a$ then $a\, mod^{+}\, b$ is equal to b.

Considering the above formulas the unranking algorithm is given in Algorithm 5.

Algorithm 4 Ranking algorithm

```
Function $Rank(Beg$: integer; $Fin$: integer)
var $R$, $Point$, $PointFin$, $Leaves$, $n$: integer;
begin
    $n := N[Beg]$;
    if $(n = 1)$ then begin
        $Fin := Beg$;
        return(1);
    end
    else begin
        $Point := Beg + 1$; $R := 0$; $Leaves := 0$;
        while $(Leaves < n\ )$ do begin
            $R := R + D[n - Leaves, N[Point] - 1]$
                $+ 2(Rank(point, PointFin) - 1) \times C[n - Leaves - N[Point]]$;
            $Leaves := Leaves + N[Point]$;
            if $N[Point] = 1$ then
                $Point := Point + 1$
            else
                $Point := PointFin + 1$;
        end;
        return$(R + 1)$;
    end;
end;
```

In this algorithm R is the input, *Beg* is the variable to show the position of the first character in the global array C and initially is set to 1. The generated codeword is hold in array C. The variable n is the number of leaves and *Root* stores the character corresponding to the current node which we want to compute the unrank of subtree rooted by this node and initially is labeled by 'm' (the label of the root).

Obviously for determining the next character we have two possibilities for the character of the root. If the root is 'r' then the next character (if exists) will be 'ℓ'. If the root is 'm' or 'ℓ' we have again two possible cases here: if all the leaves of the current tree are not produced then the next character is 'm', otherwise in this case all the leaves have been produced and then the next character will be 'r'.

Theorem 6. The time complexity of the unranking algorithm is $O(n \log n)$.

Proof. Let T be a neuronal tree with n leaves whose subtrees are defined by T_1,

Algorithm 5 Unrank algorithm

```
Function $UnRank(R, Beg, n$: integer; $Root$: char)
var $Point, i, t$: integer;
begin
    if $((n = 0)$ or $(R = 0))$ then return$(Beg - 1)$
    else
        if $(n = 1)$ then begin
            $C[Beg] := Root$;
            return$(Beg)$;
        end
        else begin
            $C[Beg] := Root$; $Point := Beg + 1$;
            $Root := \ell$;
            while $(n > 0)$ do begin
                $i := 0$;
                repeat
                    $i := i + 1$;
                until $(D[n, i] \geq R)$;
                $R := R - D[n, i - 1]$;
                if $(n - i) = 0$ then $Root := 'r'$;
                $t := 2C[n - i]$;
                $Point := UnRank((div^+(R, t)) + 1, Point, i, Root) + 1$;
                $R := mod^+(R, t)$;
                $n := n - i$; $Root := 'm'$;
            end;
            return$(Point - 1)$;
        end;
end;
```

$T_2, \ldots, T_j$ and for $1 \leq i \leq j : |T_i| = n_i$ and $\sum_{i=1}^{j} n_i = n$, and let $T(n)$ be the time complexity of unranking algorithm. With regard to the unranking algorithm, the time complexity of finding j such that $D[n, j] \geq R$ for each T_i of T is $O(\log n_i)$, therefore we have:

$$T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_j) + T(n_1) + T(n_2) + \ldots + T(n_j).$$

We want to prove that $T(n) = O(n \log(n))$. In order to obtain an upper bound for $T(n)$ we do as follows. First we prove this assumption for $j = 2$, then we generalize it. For $j = 2$ we have $T(n) = O(\log(n_1) + \log(n_2)) + T(n_1) + T(n_2)$. Let $n_1 = k$; then we can write the above formula as

$$T(n) = T(k) + T(n-k) + O(\log(k) + \log(n-k)) = T(k) + T(n-k) + C' \log(n).$$

For proving that $T(n) = O(n \log(n))$ we use an induction on n. We assume $T(m) \leq Cm \log(m)$ for all $m \leq n$, thus in $T(n)$ we can substitute

$$T(n) \leq C \times k \log(k) + C \times (n-k) \log(n-k) + C' \log(n).$$

Let $f(k) = C \times k \log(k) + C \times (n-k) \log(n-k)$, now the maximum value of $f(k)$ with respect to k and by considering n as a constant value can be obtained by evaluating the derivation of $f(k)$ which is $f'(k) = C \times \log(k) - C \times \log(n-k)$. Thus if $f'(k) = 0$ we get $k = (n-1)/2$ and by computing $f(1)$, $f(n-2)$ and $f((n-1)/2)$ we have:

$$\begin{aligned} f(1) &= f(n-2) = C \times (n-2) \log(n-2), \\ f((n-1)/2) &= 2C \times ((n-1)/2) \times \log((n-1)/2) < C \times (n-2) \log(n-2) \end{aligned}$$

so the maximum value of $(f(k))$ is equal to $C \times (n-2) \log(n-2)$ and therefore

$$T(n) \leq C \times (n-2) \log(n-2) + C' \times \log(n).$$

It is enough to assume $C = C'$; then

$$T(n) \leq C \times (n-2) \log(n) + C \times \log(n) \leq C \times n \log(n).$$

Now, for generalizing the above proof and proving $T(n) = O(n \log n)$, we should find the maximum of the function $f(n_1, n_2, \ldots, n_j) = \prod_{i=1}^{j} n_i$. By the Lagrange method we prove that maximum value of $f(n_1, n_2, \ldots, n_j)$ is equal to $\left(\frac{n}{j}\right)^j$. Then $\frac{\delta f}{\delta j} = \left(\frac{n}{j}\right)^j \left(\ln\left(\frac{n}{j}\right) - 1\right) = 0$, and

$$\begin{aligned} \ln\left(\frac{n}{j}\right) - 1 &= 0, \\ \frac{n}{j} &= e \Rightarrow j = \frac{n}{e}, \end{aligned}$$

so the maximum value of $f(n_1, n_2, \ldots, n_j)$ is equal to $e^{\frac{n}{e}}$. We know that:

$$T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_j) + T(n_1) + T(n_2) + \ldots + T(n_j),$$

so

$$\begin{aligned} T(n) &= O\left(\log\left(\prod_{i=1}^{j} n_i\right)\right) + \sum_{i=1}^{j} T(n_i), \\ T(n) &< O\left(\log\left(n^{\frac{n}{e}}\right)\right) + \sum_{i=1}^{j} T(n_i), \\ T(n) &< O\left(\frac{n}{e}\log e\right) = O(n) + \sum_{i=1}^{j} T(n_i). \end{aligned}$$

Finally by using induction, we assume that for each $k < n$ we have $T(n) < \beta n \log n$, therefore

$$\begin{aligned} T(n) &= O(n) + \sum_{i=1}^{j} T(n_i), \\ T(n) &< O(n) + \sum_{i=1}^{j} \beta O(n_i \log n_i), \\ T(n) &< O(n) + \beta \log\left(\prod_{i=1}^{j} (n_i^{n_i})\right), \\ T(n) &< O(n) + O(\log(n^n)), \\ T(n) &= O(n \log n). \end{aligned}$$

Hence, the proof is complete. □

5 APPLICATIONS

Trees have many applications in circuit design, data compression, string matching, and image processing [33, 34, 35]. For example in image processing, particular cases of t-ary trees, quadtrees and octrees, are used for the hierarchical representation of 2- and 3-dimensional images, respectively [36]. Problems of interest are the efficient construction of quadtrees and octrees from frame buffers, axial tomography scan slices computation, parametric generation, and some other techniques. Efficient encoding and decoding algorithms as well as an arithmetic which supports geometric transformations and other relevant manipulations were recently developed [36, 37]. In this representation, graphical objects can be stored and transmitted on networks using the minimum possible amount of data that is needed to rebuild the original object [38, 39].

As mentioned earlier, the class of neuronal trees is based on different biological facts. Dendrites of nerve cell [21], Darwin tree of life [40] and phylogenetic trees or evolutionary trees [26, 28, 41] modeling are some applications of neuronal trees. To show how important are they, we focus on the most important one which is phylogenetic trees.

Taxon is a group of one or more populations of organisms. For years there have been no exact criteria for what belongs or does not belong to such a taxonomic group. Today for scientists it is common to define a "good taxon" as one that reflects evolutionary or phylogenetic relationships, which directly come from evolutionary trees or phylogenetic trees [41]. By a phylogenetic tree on a set S of n taxa or species we mean a tree without out-degree one nodes whose leaves are bijectively labeled in S [28, 41]. Biologists use either unrooted (qualitative) or rooted (cladistic) evolutionary trees. On a rooted-evolutionary tree, there is a common ancestor named root that corresponds to the most ancient ancestor in the tree. Leaves of evolutionary trees correspond to the existing species (taxa) while internal vertices correspond to hypothetical ancestral species; but in the unrooted-evolutionary trees, we may not claim about the position of evolutionary ancestors or root in the tree [26, 28, 41].

There is also another classification of evolutionary trees, evolutionary trees where every internal node has exactly two children (two states) and evolutionary trees whose internal nodes have two or more children (multistate) [26]. So the rooted-multistate-evolutionary trees of n species is equivalent to the neuronal trees of size n when the rooted-two states-evolutionary trees of n species are equivalent to binary trees.

In the past, biologists relied on morphological features, like the number of legs (for insects) or beak shapes (for birds) or the presence or absence of fins (for fishes) to construct evolutionary trees; but today biologists use DNA sequences information for reconstruction of evolutionary trees [42]. Considering a set of n species, each species has some special information corresponding to its DNA, like genes. Biologists prefer to build evolutionary trees based on these information (let's call these genes or information "characters"). So, biologists prefer to reconstruct evolutionary tree based on a given $n \times m$ alignment matrix (n species and m characters); we call this approach "character-based tree reconstruction method" [43, 44].

An intuitive score for a character-based evolutionary tree is the total number of mutations (changing one character form a parent to a child) required to explain all of the observed character sequences [45, 46]. The "parsimony approach" attempts to minimize this score. Practically, the goal is to find the strings of characters assigned to internal vertices when minimizing the parsimony score. This problem is called "large parsimony problem" which assumes that neither the tree structure nor the labels of its internal vertices are known and tries to minimize the parsimony score [43, 47].

However, the number of topologies grows very fast with respect to n; to find the best solution we need to search all possible topologies of evolutionary trees, so biologists often use local heuristic searches or approximation algorithms. We proposed a constant average time complexity algorithm to generate all neuronal trees of the same size. As these trees are equivalent to evolutionary trees, using this method enables us to efficiently generate all topologies of evolutionary trees of the same size (same number of taxa) for finding the one with the best score.

This work comes with all other trespassory algorithms like ranking, unranking and complexity analysis which show that these algorithms are efficient and easy to store, generate and use practically.

6 CONCLUSION

In this paper, we presented an efficient algorithm for the generation of neuronal trees with n leaves in A-order with an encoding over three letters (an encoding with minimum size of alphabet such that the length of each codeword is less than $2n$). Also, two genuine ranking and unranking algorithms were designed for this encoding. The generation algorithm has $O(n)$ time complexity in the worst case and $O(1)$ in an average case. The ranking and unranking algorithms have O(n) and $O(n \log n)$ time complexity, respectively.

Acknowledgment

This research was in part supported by a grant from IPM (No. CS1386-4-08) and University of Tehran. The authors would like to thank the anonymous referees for their helpful suggestions.

REFERENCES

[1] Pallo, J.—Racca, R.: A Note on Generating Binary Trees in A-Order and B-Order. International Journal of Computer Mathematics, Vol. 18, 1985, pp. 27–39.

[2] Lucas, J.—Roelants van Baronaigien, D.—Ruskey, F.: On Rotations and the Generation of Binary Trees. Journal of Algorithms, Vol. 15, 1993, pp. 343–366.

[3] Vajnovszki, V.—Pallo, J.: Generating Binary Trees in A-Order from Codewords Defined on Four-Letter Alphabet. Journal of Information and Optimization Science, Vol. 15, 1994, pp. 345–357.

[4] Ahrabian, H.—Nowzari-Dalini, A.: On the Generation of Binary Trees from (0-1) Codes. International Journal of Computer Mathematics, Vol. 69, 1998, pp. 243–251.

[5] Ahrabian, H.—Nowzari-Dalini, A.: On the Generation of Binary Trees in A-Order. International Journal of Computer Mathematics, Vol. 71, 1999, pp. 351–357.

[6] Ahrabian, H.—Nowzari-Dalini, A.: Parallel Generation of Binary Trees in A-Order. Parallel Computing, Vol. 31, 2005, pp. 948–955.

[7] Wu, R.—Chang, J.—Wang, Y.: A Linear Time Algorithm for Binary Tree Sequences Transformation Using Left-Arm and Right-Arm Rotations. Theoretical Computer Science, Vol. 335, 2006, pp. 303–314.

[8] Ruskey, F.: Generating t-ary Trees Lexicographically. SIAM Journal of Computing, Vol. 7, 1978, pp. 424–439.

[9] Er, M. C.: Efficient Generation of k-ary Trees in Natural Order. Computer Journal, Vol. 35, 1992, pp. 306–308.

[10] Vajnovszki, V.—Pallo, J.: Ranking and Unranking of k-ary Trees with $4k-4$ Letter Alphabet. Journal of Information and Optimization Science, Vol. 18, 1997, pp. 271–279.

[11] Korsh, J. F.—LaFollette, P.: Loopless Generation of Gray Codes for k-ary Trees. Information Processing Letters, Vol. 70, 1999, pp. 7–11.

[12] Xiang, L.—Ushijima, K.—Tang, C.: On Generating k-ary Trees in Computer Representation. Information Processing Letters, Vol. 77, 2001, pp. 231–238.

[13] Korsh, J. F.: Generating t-ary Trees in Linked Representation. Computer Journal, Vol. 48, 2005, pp. 488–497.

[14] Ahrabian, H.—Nowzari-Dalini, A.: Parallel Generation of t-ary Trees in A-Order. Computer Journal, Vol. 50, 2007, pp. 581–588.

[15] Heubach, S.—Li, N.—Mansour, T.: Staircase Tilings and k-Catalan Structures. Discrete Mathematics, Vol. 308, 2008, pp. 5954–5964.

[16] Manes, K.—Sapounakis, A.—Tasoulas, I.—Tsikouras, P.: Recursive Generation of k-ary Trees. Journal of Integer Sequences, Vol. 12, 2009, pp. 1–18.

[17] Ahmadi-Adl, A.—Nowzari-Dalini, A.—Ahrabian, H.: Ranking and Unranking Algorithms for Loopless Generation of t-ary Trees. Logic Journal of IGPL, Vol. 19, 2011, pp. 33–43.

[18] Wu, R.—Chang, J.—Chang, C.: Ranking and Unranking of Non-Regular Trees with a Prescribed Branching Sequence. Mathematical and Computer Modelling, Vol. 53, 2011, pp. 1331–1335.

[19] Pallo, J.: Generating Trees with n Nodes and m Leaves. International Journal of Computer Mathematics, Vol. 21, 1987, pp. 133–144.

[20] Seyedi-Tabari, E.—Ahrabian, H.—Nowzari-Dalini, A.: A New Algorithm for Generation of Different Types of RNA. International Journal of Computer Mathematics, Vol. 87, 2010, pp. 1197–1207.

[21] Pallo, J.: A Simple Algorithm for Generating Neuronal Dendritic Trees. Computer Methods and Programs in Biomedicine, Vol. 33, 1990, pp. 165–169.

[22] Vajnovszki, V.: Listing and Random Generation of Neuronal Trees Coded by Six Letters. The Automation, Computers, and Applied Mathematics, Vol. 4, 1995, pp. 29–40.

[23] Li, L.: Ranking and Unranking AVL Trees. SIAM Journal of Computing, Vol. 15, 1986, pp. 1025–1035.

[24] Shin, J.—Choi, D.: Search Tree Generation for the Exception Handling of E-Commerce Delivery Process. Computing and Informatics, Vol. 30, 2011, pp. 733–747.

[25] Zaks, S.: Lexicographic Generation of Ordered Trees. Theoretical Computer Science, Vol. 10, 1980, pp. 63–82.

[26] Day, W. H. E.: The Computational Complexity of Inferring Rooted Phylogenies by Parsimony. Mathematical Biosciences, Vol. 81, 1986, pp. 33–42.

[27] Mir, A.—Rosselló, F.: The Mean Value of the Squared Path-Difference Distance for Rooted Phylogenetic Trees. Journal of Mathematical Analysis and Applications, Vol. 371, 2010, pp. 168–176.

[28] Alberich, R.—Cardona, G.—Rosselló, F.—Valiente, G.: An Algebraic Metric for Phylogenetic Trees. Applied Mathematics Letters, Vol. 22, 2009, pp. 1320–1324.

[29] Sloane, N. J. A.: A Handbook of Integer Sequences. Academic Press, NY, 1973.

[30] Cameron, P.: Some Sequences of Integers. Discrete Mathematics, Vol. 75, 1989, pp. 89–102.

[31] Bonin, L. S. J.—Simion, R.: Some q-Analogues of the Schröder Numbers Arising from Combinatorial Statistics on Lattice Paths. Journal of Statistical Planning and Inference, Vol. 34, 1993, pp. 35–55.

[32] Coker, C.: A Family of Eigensequences. Discrete Mathematics, Vol. 282, 2004, pp. 249–250.

[33] Knuth, D. E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 2^{nd} Ed., Addison-Wesley, Reading, MA, 1973.

[34] Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA, 1989.

[35] Steinberger, J.—Ježek, K.: Evaluation Measures for Text Summarization. Computing and Informatics, Vol. 28, 2009, pp. 251–275.

[36] Samet, H.—Webber, R. E.: Hierarchical Data Structures and Algorithms for Computer Graphics. IEEE Computer Graphs & Applications, Vol. 8, 1988, pp. 67–75.

[37] Wilke, L. M.—Schrack, G. F.: Improved Mirroring and Rotation Functions for Linear Quadtree Leaves. Image and Vision Computing, Vol. 13, 1995, pp. 491–495.

[38] Samet, H.—Kochut, A.: Octree Approximation and Compression Methods. In Proceedings of the First International Symposium on 3D Data Processing Visualization and Transmission, Padova, Italy, June 2002, IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 460–469.

[39] Stewart, I. P.: Quadtrees: Storage and Scan Conversion. Computer Journal, Vol. 29, 1986, pp. 60–75.

[40] Darwin, C: On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life. John Murray, London, UK, 1859.

[41] Queiroz, K. D. —Gauthier, J.: Phylogeny as a Central Principle in Taxonomy: Phylogenetic Definitions of Taxon Names. Systematic Zoology, Vol. 39, 1990, pp. 307–322.

[42] Gómez, J. M.—Fuentes Lorenzo, D.—García Crespo, Á.—Han, S.-K.: SE-BIO: A Semantic Bioinformatics Platform for the New E-Science. Computing and Informatics, Vol. 27, 2008, pp. 37–52.

[43] Jone, N. C.—Pevzner, P. A.: An Introduction to Bioinformatics Algorithms. The MIT Press, Massachusetts, 2004.

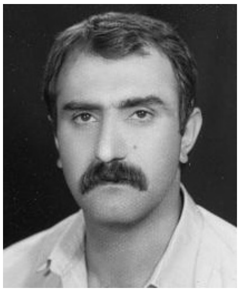
[44] Takahashi, K.—Nei, M.: Efficiencies of Fast Algorithms of Phylogenetic Inference Under the Criteria of Maximum Parsimony, Minimum Evolution, and Maximum Likelihood When a Large Number of Sequences Are Used. Molecular Biology and Evolution, Vol. 17, 2000, pp. 1251–1258.

[45] Fitch, W. M.: Toward Defining the Course of Evolution: Minimum Change for a Specific Tree Topology. Systematic Zoology, Vol. 20, 1971, pp. 406–416.

[46] Robinson, D. F.: Comparison of Labeled Trees with Valency Three. Journal of Combinatorial Theory (B), Vol. 11, 1971, pp. 105–119.

[47] Kolaczkowski, B.—Thornton, J. W.: Performance of Maximum Parsimony and Likelihood Phylogenetics When Evolution Is Heterogeneous. Nature, Vol. 431, 2004, pp. 980–984.

Mahdi Amani is a Ph. D. student in the Department of Computer Science, School of Mathematics, Statistics and Computer Science, University of Tehran, under supervision of Dr. A. Nowzari. His research interests include combinatorial algorithms, graph theory, and bioinformatics.



Abbas Nowzari-Dalini is an Associate Professor of School of Mathematics, Statistics and Computer Science, University of Tehran. He is currently Head of Department of Computer Science at this school. His research interests include combinatorial algorithms, parallel algorithms, DNA computing, bioinformatics, neural networks, and computer networks.

Hayadeh Ahrabian was a Full Professor of School of Mathematics, Statistics and Computer Science, University of Tehran. Her research interests included combinatorial algorithms, parallel algorithms, DNA computing, bioinformatics, and genetic algorithms. This work is her final work, and unfortunately, after a long struggle against cancer, she passed away before the publication of this paper.