Abstract — In this paper, we present two new ranking and unranking algorithms for $k$-ary trees represented by $x$-sequences in Gray code order. These algorithms are based on a gray code generation algorithm developed by Ahrabian et al. In mentioned paper, a recursive backtracking generation algorithm for $x$-sequences corresponding to $k$-ary trees in Gray code was presented. This generation algorithm is based on Vajnovszki’s algorithm for generating binary trees in Gray code ordering. Up to our knowledge no ranking and unranking algorithms were given for $x$-sequences in this ordering. We present ranking and unranking algorithms with $O(kn^2)$ time complexity for $x$-sequences in this Gray code ordering.

Keywords — $k$-ary Tree Generation, Ranking, Unranking, Gray Code.

I. INTRODUCTION

Trees are one of the most important structures in computer science. Trees have many applications such as database generation, decision table programming, analysis algorithms, string matching [11], switching theory, and even in the theoretical VLSI circuit design [26]. Trees are also widely used data structures for maintaining data [19] and as auxiliary structures for compressing data [10]. In addition, the exhaustive generation of all trees of a certain type is often useful; for example, a list of all trees with a given number of nodes $n$, may be used to test and analyze of algorithm complexity, and prove the correctness of an algorithm [11]. As such, the problem of generating binary trees [1], [4], [17], [21], [22], [30], $k$-ary trees [2], [5], [6], [7], [8], [13], [18], [23], [28], and other types of trees [20], [25], has been thoroughly investigated and many papers have been published in the literature for generating trees.

In most of these algorithms, trees are encoded as integer sequences and then these sequences are generated with a certain order, and consequently their corresponding trees are also generated in a specific order. The most well-known orderings on trees are A-order and B-order [22], [33], and the orderings on the sequences are lexicographical [33] and Gray code [15], [17]. Typically, an important issue for algorithms to enumerate various classes of objects is that they must be run in a constant time for each generation. Therefore so many algorithms have been proposed to generate $k$-ary trees in constant time in the worst case. These algorithms generate the sequences corresponding to trees in Gray code order [14], [15], [16], [24], [32].

Beside the generation algorithm for $k$-ary trees, ranking and unranking algorithms are also important in the concept of generation [23], [28], [31], [33]. Given a specific order on the set of $k$-ary trees, the rank of a tree (or corresponding sequence) is its position in the exhaustive generated list, and the ranking algorithm computes the rank of a given tree (or corresponding sequence) in this order. The reverse operation of the ranking is called unranking, which generates the tree (or sequence) corresponding to a given rank. Usually tree generation papers present ranking and unranking algorithms for tree generation algorithm. Ranking and unranking algorithms also have many applications. For example, in traditional tree compression algorithm for encoding the tree to code sequence and decoding the code sequence back to a tree, the ranking and unranking algorithms can be used.

As mentioned, in tree generation algorithms, trees are represented as sequences and then these sequences are generated. Two well-known representations of $k$-ary trees with $n$ nodes are $x$-sequences (or 0-1 sequences) and $z$-sequences presented by Zaks [33]. He presented a generation algorithm for $z$-sequences encoding in lexicographical ordering such that the corresponding trees are generated in B-order, and generates one sequence from a given sequence in $O(n)$ ($n$ is the length of $z$-sequence). He also presented two ranking and unranking algorithms for this generation in $O(kn)$. Later, some other $k$-ary tree generation algorithms were presented on this encoding [7], [24], [32].

Vajnovszki presented a new Gray code for the $x$-sequence corresponding to binary trees with $n$ nodes [27]. A constant average time algorithm for generating this Gray code is also given. Ahrabian et al. introduced another algorithm for generating $k$-ary trees represented by $x$-sequence in Gray code order [3], which is based on Vajnovszki’s paper [27]. This algorithm is a backtracking generation algorithm. Up to now, no ranking and unranking algorithms are presented for $x$-sequences in Gray code ordering given by Vajnovszki [27] and Ahrabian et al. [3].

In this paper, we present ranking and unranking algorithms for $x$-sequences corresponding to $k$-ary trees with $n$ nodes generated in Gray code based on the Ahrabian et al. [3] generation algorithm. The time complexity of ranking algorithm is $O(kn^2)$ and unranking algorithm is also $O(kn^2)$.

The rest of this paper is organized as follows. Section II introduces the definitions and notations that are used subsequently. In Section III, ranking and unranking algorithms are discussed. Finally, some concluding remarks are offered in Section IV.
II. PRELIMINARIES

In graph theory, a tree is a connected and undirected graph without any cycle. A rooted tree is a tree with a distinct node which is called root. Rooted trees are directed trees and the direction is from the root to the offspring. A rooted tree where the children of each node have a designated order is called ordered tree. A k-regular ordered tree is an ordered tree in which each node has either k children (internal node) or no children (external node or leaf), and is called k-ary tree. An n-node k-ary tree has n internal nodes and (k−1)n+1 external nodes or leaves, and the total number of k-ary trees with n internal nodes is denoted by Cn,k and is known to have the value

\[ C_{n,k} = \frac{(kn)}{(k-1)n+1} \]

A k-ary tree T can also be defined recursively as being either an external (leaf) or an internal node altogether with a sequence T1, T2, ..., Tk of k-ary trees, which Ti is defined as the ith subtree of T [11].

Let us introduce basic notations used through this paper and a definition of k-ary trees by means of choice functions of indexed families of sets (from [9], [12], [13]).

Let \( \Gamma = \{1, \ldots, m\} \) and \( I = \{1, \ldots, \ell\} \) be two sets, and \( \ell, m \geq 1 \), then \( \Gamma_i \supseteq I \) where \( \Gamma_i = \Gamma \), show an family of sets indexed by I. Any mapping function \( f \) which chooses one element from each set \( \Gamma_1, \ldots, \Gamma_\ell \) is called a choice function of the indexed family \( \Gamma_i > I \). With supplementary restrictions, various classes of combinatorial objects can be modeled by choice functions [9], [12], [13].

If \( I = \{1, \ldots, \ell\} \) and \( \Gamma_i = \{0, 1\} \), then any choice function \( \chi = \chi_{1,i} \) that belongs to the indexed family \( \Gamma_i > I \), is called binary choice function of this family [12]. If \( \ell \leq kn \) for a given \( k \) and \( n \), each binary choice function which \( x_1 + \cdots + x_i \geq i/k \), for \( 1 \leq i \leq kn \), is called binary choice function with \( k \)-dominating properties. There exist bijections between set of choice functions \( \chi \) and sets of k-ary trees with \( n \) internal nodes in widely used representations. All \( k \)-dominating binary choice functions, with \( \ell = kn \) and the number of \( x_1 + \cdots + x_i = n \), are bitstring representations of all k-ary trees of the set \( \Gamma \). This bitstring representation is called x-sequence [33]. By \( k \)-dominating definition, in each subsequence \( \{x_i\}_{i=1}^{k} \) \( (1 \leq i \leq kn) \) the accumulated number of 1’s is at least \( \lceil i/k \rceil \). Clearly, each sequence has \( n \) 1’s and \( (k-1)n \) 0’s.

For any given choice functions \( \delta = d_1, \ldots, d_\ell \) and \( \gamma = g_1, \ldots, g_\ell \), we say that \( \delta \) and \( \gamma \) are in Gray code order, if they differ by a constant number of changes [29].

The x-sequence can be obtained directly from k-ary trees. Given a regular k-ary tree with \( n \) internal nodes, we label each internal node with 1 and each external node with 0. Reading tree labels in pre-order (recursively, visit first root and then all the subtrees from left to right), we get a bitstring with \( n \) 1’s and \( (k-1)n + 1 \) 0’s. As the last visited node is an external node, we omit the corresponding 0 [33]. For example, the x-sequence corresponding to the tree presented in Fig. 1 is \( x = 1100001000100 \).

Now, let \( S_{k,n} \) be a set of all \( k \)-dominating binary choice functions with the number of \( x_1 + \cdots + x_i = n \) and \( m = \ell - n \), in other words the set of all bitstrings with a \( k \)-dominating property with \( m \) 0’s and \( n \) 1’s such that \( m \geq (k-1)n \). In particular case where \( m = (k-1)n \), \( S_{k,n} \) is the set of all x-sequences of length \( kn \). As it is mentioned, we call a list of bitstrings in \( S_{k,n} \) Gray code list if every two successive bitstrings in the list differ by an interchange of two bits. Also, let \( \alpha \) and \( \beta \) be two bitstrings then we denote \( \alpha \beta \) the concatenation of \( \alpha \) and \( \beta \). By these definitions, the following lemma shows the recursion property of the set \( S_{k,n} \), which helps us in the generation schema [3].

**Lemma 1.** For any \( k > 0 \), and \( m, n \geq 0 \), we have

\[
S_{k,n} = \begin{cases} 
0^n & \text{if } n = 0, \\
S_{k,n-1}^{m-1} & \text{if } m = (k-1)n, \\
S_{k,n-1}^{m,n-1} & \text{if } m > (k-1)n > 0.
\end{cases}
\]

For example, Table 1 shows all bitstrings \( S_3^{6,2} = 0S_3^{5,2} \cup 1S_3^{6,1} \). With regard to the recursion property of the set \( S_{k,n} \), we can easily count the number of elements in this set. Let \( S_{k,n} \) be the number of bitstrings in \( S_{k,n} \). According to the above lemma, \( S_{k,n} \) can be obtained as follows.

**Corollary 1.** For any \( k > 0 \), and \( m, n \geq 0 \), we have

\[
S_{k,n} = \begin{cases} 
1 & \text{if } n = 0, \\
S_{k,n}^{m-1} & \text{if } m = (k-1)n, \\
S_{k,n}^{m,n-1} + S_{k,n-1}^{m,n-1} & \text{if } m > (k-1)n > 0.
\end{cases}
\]

![Fig. 1. A 3-ary tree with 4 internal nodes.](image)

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>ALL BITSTRINGS IN S_{3}^{6,2} = 0S_{3}^{5,2} \cup 1S_{3}^{6,1}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0S_{3}^{5,2}</td>
<td>1S_{3}^{6,1}</td>
</tr>
<tr>
<td>00100100</td>
<td>1100000</td>
</tr>
<tr>
<td>00101000</td>
<td>10100000</td>
</tr>
<tr>
<td>00110000</td>
<td>10010000</td>
</tr>
<tr>
<td>01000000</td>
<td>10001000</td>
</tr>
<tr>
<td>01100000</td>
<td>10000100</td>
</tr>
<tr>
<td>01010000</td>
<td>10000100</td>
</tr>
<tr>
<td>10000100</td>
<td>1</td>
</tr>
</tbody>
</table>
Corollary 2. For any \( k > 0, n \geq 0, \) and \( m \geq (k - 1)n \geq 0, \) we have

\[
S_k^{m,n} = \frac{m + 1 - (k - 1)n}{m + 1} \binom{m + n}{n}.
\]

With regard to the definition of \( S_k^{m,n} \), it should be noted that \( S_k^{(k-1)n,n} \) is the set of bitstrings corresponding to \( k \)-ary trees with \( n \) internal nodes. Therefore, we can have the following theorem.

Theorem 1. In particular case of Corollary 2 where \( m = (k - 1)n \),

\[
S_k^{(k-1)n,n} = \frac{1}{(k - 1)n + 1} \binom{kn}{n}.
\]

Obviously \( S_k^{(k-1)n,n} = C_{n,k} \) (the number of all \( k \)-ary trees with \( n \) nodes). With regard to the above discussion and using Lemma 1 the generation \( k \)-ary trees algorithm in Gray code order was presented in [3]. As an example, a list of 22 \( x \)-sequences corresponding to 4-ary trees with \( n = 3 \) nodes in this Gray code order is shown in Table II.

III. RANKING AND UNRANKING ALGORITHM

The rank of a \( k \)-ary tree with respect to some ordering is the number of previously generated trees in that ordering. The reverse operation of the ranking is unranking. As mentioned, for the \( x \)-sequences in Gray code, no ranking and unranking algorithms are presented in the literature in Gray code ordering given by Vajnovszki [27] and Ahrabian et al. [3]. In this section, the ranking and unranking algorithms for these generation and ordering are presented.

For ranking and unranking algorithms we need \( S_k^{m,n} \) for given \( k \), \( m \), and \( n \), that are defined and computed earlier. We assume that these values are computed and stored in arrays \( S[m][n] \) for a fixed \( k \). Also, for computing the rank of a given bitstring or unranking a given rank, we need three other variables \( \text{prePos}, \text{curPos}, \text{and dir} \). In the ranking and unranking algorithm, when the \( i \)-th of a bitstring is investigated, the variable \( \text{curPos} \) holds the position of this \( i \)-th \( 1 \) and the variable \( \text{prePos} \) holds the position of previous \( 1 \) (\( i - 1 \)-th \( 1 \)).

The variable \( \text{dir} \) holds the direction of the \( i \)-th \( 1 \) in a bitstring. Direction of the \( i \)-th \( 1 \) in a bitstring is \( \text{down (represented by 0)} \) if the \( i \)-th \( 1 \) has been shifted one place to the left in the \( i \)-th \( 1 \) and in Gray code order, and if the \( i \)-th \( 1 \) has been shifted one place to the right, the direction is \( \text{up (represented by 1)} \). For example, consider two bitstrings ‘00100100’ and ‘00100010’, for the first bitstring the direction of the second \( 1 \) is \( \text{down} \), because in the next bitstring, the second \( 1 \) has been shifted to the left. But for two bitstrings ‘01010000’ and ‘01011000’, the direction of second \( 1 \) in the first bitstring is \( \text{up} \), because this \( 1 \) has been shifted to right. Since the first bit in the input bitstring is always equal to \( 1 \) and does not shift, we can assume that the direction of the first bit is equal to 0.

To determine the direction of the \( i \)-th \( 1 \) for \( 2 \leq i \leq n \), we can use the following relation:

If direction of the \( (i - 1) \)-th \( 1 \) is equal to 0 then direction of the \( i \)-th \( 1 \) is equal to

\[
\begin{cases}
0 & \text{if } (i - 1) \text{-th } 1 \text{ is in its last position,} \\
1 & \text{if } (i - 1) \text{-th } 1 \text{ is not in its last position.}
\end{cases}
\]

If direction of the \( (i - 1) \)-th \( 1 \) is equal to 1 then direction of the \( i \)-th \( 1 \) is equal to

\[
\begin{cases}
1 & \text{if } (i - 1) \text{-th } 1 \text{ is in its last position,} \\
0 & \text{if } (i - 1) \text{-th } 1 \text{ is not in its last position.}
\end{cases}
\]

In this relation, the position of the \( i \)-th \( 1 \) is equal to \( (i - 1)k + 1 \), then it is in the last position.

As mentioned, a \( x \)-sequence corresponding to a \( k \)-ary tree with \( n \) nodes, has \( n \) \( 1 \)'s and \( m = (k - 1)n \) \( 0 \)'s. In the Gray code order, for generating a bitstring from the previous bitstring, each \( 1 \) should be shifted from its position to next position. Therefore, each \( 1 \) has a shift direction and can be computed using the above relation. If the position of the \( i \)-th \( 1 \) in the \( j \)-th bitstring is \( p \) and its direction is \( \text{up (dir is equal to 1)} \) and this \( 1 \) is a candidate for shifting, then its position in the \( (j + 1) \)-th bitstring is \( p + 1 \). Otherwise, if the direction of the \( i \)-th \( 1 \) in the \( j \)-th bitstring is \( \text{down (dir is equal to 0)} \) and this \( 1 \) is a candidate for shifting, then its position in the \( (j + 1) \)-th bitstring is \( p - 1 \). A \( 1 \) is shiftable if it is not in its last or first position, in other words, a \( 1 \) is shiftable if its shift direction is \( \text{up} \) and it is not in its last position or, if its shift direction is \( \text{down} \) and it is not in its first position.

In order to compute the rank of a bitstring, we count the number of generated bitstrings before this bitstring in the Gray code order. This can be done, using by the shiftable \( 1 \)'s. For this purpose, the bitstring should be scanned from left to right and a shiftable \( 1 \) should be found. As mentioned, the first \( 1 \) is never shifted and is always in position 1, and shift direction of \( 1 \)-th \( 1 \) is determined by shift direction of \( (i - 1) \)-th \( 1 \). Assume that we investigate the \( i \)-th \( 1 \) in the \( j \)-th bitstring, we have two cases which are explained as follows.

In the first case, the shift direction of the \( i \)-th \( 1 \) in the \( j \)-th bitstring is equal to \( \text{down (dir = 0)} \). So, the first possible position is \( p_{\text{last}} \) for this \( 1 \), where is equal to \( (i - 1)k + 1 \) and the last possible position is \( p_{\text{first}} \) for this \( 1 \), where is equal to the position of the \( (i - 1) \)-th \( 1 \) plus 1. In all the consecutive bitstrings that are generated by shifting of the \( i \)-th, the possible positions for this \( 1 \) can be the positions \( p_{\text{last}} - 1, p_{\text{last}} - 2, \ldots, p_{\text{first}} \). Therefore, if the position of the \( i \)-th \( 1 \) is \( p \) then the positions of this \( 1 \) in previous bitstrings are \( p_{\text{last}} - 1, \ldots, p - 2, p + 1 \). The number of such bitstrings should be obtained for ranking. At first, we count the number of bitstrings which the \( i \)-th \( 1 \) of them is in position \( p_{\text{last}} \). The positions of the first 1 to \( (i - 1) \)-th 1 are fixed and we have to count the number of 0’s and 1’s which their positions are after the position of \( i \)-th 1. The number of 1’s that their positions are after the \( i \)-th 1 in the \( j \)-th bitstring is equal to \( n'_{i} = n - i \). Now
let $q$ be the number of $0$’s which their positions are before the $i$th 1 in the $j$th bitsring, so the number of $0$’s which their positions are after $p$ is equal to $m’_i = m - q$ (it is obvious that $q = p - i$). Therefore, the number of bitsrings that the $i$th 1 of them is in position $p_{\text{last}}$ is equal to $S^{m’_i, n’_j}_k$. With the same way, we can count the number of bitsrings which the $i$th 1 of them are in the positions $p_{\text{last}} - 1, \cdots, p + 2$, $p + 1$. Finally, the number of previous bitsrings before bitsring $j$ that the $i$th 1 of them are in positions $p_{\text{last}}, p_{\text{last}} - 1, \cdots, p + 2$, $p + 1$ is equal to:

$$\sum_{j=p+1}^{p_{\text{last}}} S^{m’_i, n’_j}_k.$$ 

In the second case, the shift direction of the $i$th 1 in the $j$th bitsring is equal to $u$ ($\text{dir} = 1$). So the first possible position for this 1 is $p_{\text{first}}$. In all the consecutive bitsrings which are generated by shifting of the $i$th 1, the possible positions for this 1 can be $p_{\text{first}}, p_{\text{first}} + 1, p_{\text{first}} + 2, \cdots, p_{\text{last}}$. Therefore, if the position of the $i$th 1 is $p$ then the positions of this 1 in previous bitsrings are $p_{\text{first}}, p_{\text{first}} + 1, \cdots, p - 2$, $p - 1$. The number of such bitsrings should be obtained for ranking. This number can be computed similar to the first case and we have:

$$\sum_{j=p_{\text{first}}}^{p-1} S^{m’_i, n’_j}_k,$$

where the values of $m’_j$ and $n’_j$ are equal to $m - q$ and $n - i$, respectively (similar to the first case).

Now, the rank of a given bitsring can be computed as:

$$\sum_{i=2}^{n} \sum_{j=p_{\text{first}}}^{p_{\text{last}}} S^{m’_i, n’_j}_k,$$

where $p_{\text{first}}$ and $p_{\text{last}}$ are lower and upper bound for positions of the $i$th 1.

With regards to the above discussion, the ranking algorithm is given in Algorithm 1. This algorithm takes $n$, $k$, and bitsring $x$ corresponding to the $k$-ary tree with $n$ nodes as input parameters, and returns the rank of $x$ in variable $r$. Considering the above discussion, the time complexity of the ranking algorithm is $O(kn^2)$.

The unranking algorithm essentially reverses the steps carried out in computing the rank. The unranking algorithm given in Algorithm 2 takes $r$, $k$, and $n$ as input, and returns the $x$-sequence corresponding to the $k$-ary tree with $n$ nodes that has rank $r$.

For a given $r$ as a rank, the corresponding bitsring $x = x_1 \cdots x_{kn}$ is computed by the following operations. Initially $x_1$ is set to 1 and all $x_i$’s $(2 \leq i \leq kn)$ are set to zero, the position of the first 1 is set to 1, and the direction of this 1 is set to 0. The bitsring $x$ has to have $n$ 1’s, therefore the correct position of the $i$th 1 $(2 \leq i \leq n)$ is computed in a loop. In the iteration $i$ $(2 \leq i \leq n)$ of this loop, at first, the direction of the $i$th 1 $(\text{dir})$ is computed using by the direction and position of the $(i - 1)$th 1, after that, the position of the $i$th 1 $(\text{curPos})$ is computed using $s[n*(k-1) - \text{curPos} + i][n-i]$. The values of $s[n*(k-1) - \text{curPos} + i][n-i]$ determine the possible number of shifts which $i$th 1 can have. Finally, the value of this position is set to 1 in bitsring $x$. Considering the above discussion, the time complexity of the unranking algorithm is $O(kn^2)$.

IV. CONCLUSION

we have introduced two new ranking and unranking algorithms for $n$-node $k$-ary trees represented by $x$-sequences. In these algorithms, it is assumed that the order of $x$-sequences corresponding to the list of all $k$-ary trees with $n$ nodes is Gray code order. The time complexity of both ranking and unranking algorithms is $O(kn^2)$.

REFERENCES

Algorithm 1 The ranking algorithm.

Procedure Ranking(x : bitstring; n, k : integer)
begin
    r = 1;
    curPos = 1; dir = 0;
    for i = 2 to n do begin
        prePos = curPos;
        curPos = curPos + 1;
        while (x[curPos] = 1)
            curPos = curPos + 1;
        if ((dir == 0) && (prePos == (i - 2) * k + 1)) ||
            (dir == 1) && (prePos! = (i - 2) * k + 1)) then begin
            dir = 0;
            for j = (i - 1) * k + 1 downto curPos + 1 do
                r = r + s[(k - 1) * n - j + i][n - i];
        end
    else begin
        dir = 1;
        for j = prePos + 1 to curPos - 1 do
            r = r + s[(k - 1) * n - j + i][n - i];
    end
end
return r;
end

Algorithm 2 The unranking algorithm.

Procedure UnRanking(r, n, k : integer)
begin
    x[1] = 1;
    for i = 2 to kn do
        x[i] = 0;
    prePos = 1; dir = 0;
    for i = 2 to n do begin
        if ((dir == 0) && (prePos == (i - 2) * k + 1)) ||
            (dir == 1) && (prePos! = (i - 2) * k + 1)) then begin
            dir = 0;
            curPos = (i - 1) * k + 1;
            while r > s[n * (k - 1) - curPos + i][n - i] do begin
                r = r - s[n * (k - 1) - curPos + i][n - i];
                curPos = curPos + 1;
            end
        end
    else begin
        dir = 1;
        curPos = prePos + 1;
        while r > s[n * (k - 1) - curPos + i][n - i] do begin
            r = r - s[n * (k - 1) - curPos + i][n - i];
            curPos = curPos + 1;
        end
    end
    x[curPos] = 1;
    prePos = curPos;
end
return x;
end


