A Bijective Code for k-Trees with Linear Time Encoding and Decoding

Saverio Caminiti[1], Emanuele G. Fusco[1], and Rossella Petreschi[1]

Computer Science Department
University of Rome "La Sapienza", via Salaria, 113-00198 Rome, Italy
{caminiti, fusco, petreschi}@di.uniroma1.it

Abstract. The problem of coding labeled trees has been widely studied in the literature and several bijective codes that realize associations between labeled trees and sequences of labels have been presented. k-trees are one of the most natural and interesting generalizations of trees and there is considerable interest in developing efficient tools to manipulate this class, since many NP-Complete problems have been shown to be polynomially solvable on k-trees and partial k-trees. In 1970 Rényi and Rényi generalized the Prüfer code to a subset of labeled k-trees; subsequently, non redundant codes that realize bijection between k-trees (or Rényi k-trees) and a well defined set of strings were produced. In this paper we introduce a new bijective code for labeled k-trees which, to the best of our knowledge, produces the first encoding and decoding algorithms running in linear time with respect to the size of the k-tree.

1 Introduction

The problem of coding labeled trees, also called Cayley's trees after the celebrated Cayley's theorem [6], has been widely studied in the literature. Coding labeled trees by means of strings of vertex labels is an interesting alternative to the usual representations of tree data structures in computer memories, and it has many practical applications (e.g. Evolutionary algorithms over trees, random trees generation, data compression, and computation of forest volumes of graphs). Several different bijective codes that realize associations between labeled trees and strings of labels have been introduced, see for example [7, 9, 10, 17–20]. From an algorithmic point of view, the problem has been investigated thoroughly and optimal encoding and decoding algorithms are known for most of these codes [4, 5, 7, 9, 19].

k-trees are one of the most natural and interesting generalizations of trees (for a formal definition see Section 2) and there is considerable interest in developing efficient tools to manipulate this class of graphs. Indeed each graph with treewidth k is a subgraph of a k-tree, and many NP-Complete Problems (e.g. Vertex Cover, Graph k-Colorability, Independent Set, Hamiltonian Circuit, etc.) have been shown to be polynomially solvable when restricted to graphs of bounded treewidth. We suggest the interested reader to see [2, 3].

In 1970 Rényi and Rényi [21] generalized Prüfer's bijective proof of Cayley's theorem to code a subset of labeled k-trees (Rényi k-trees). They introduced a

redundant Prüfer code for Rényi k-trees and then characterized the valid codewords. Subsequently, non redundant codes that realize bijection between k-trees (or Rényi k-trees) and a well defined set of strings were produced [8, 11] together with encoding and decoding algorithms. Attempts have been made to obtain an algorithm with linear running time for the redundant Prüfer code [15], however to the best of our knowledge, no one has developed linear time algorithms for non redundant codes.

In this paper we present a new bijective code for k-trees, together with encoding and decoding algorithms, whose running time is linear with respect to the size of the encoded k-tree.

2 Preliminaries

In this section we recall the concepts of k-trees and Rényi k-trees and highlight some properties related to this class of graphs.

Let us call q-clique a clique on q nodes and $[a, b]$ the interval of integer from a to b (a and b included).

Definition 1. *[14] An unrooted k-tree is defined in the following recursive way:*

1. *A complete graph on k nodes is a k-tree.*
2. *If $T'_k = (V, E)$ is a k-tree, $K \subseteq V$ is a k-clique and $v \notin V$, then $T_k = (V \cup \{v\}, E \cup \{(v, x) \mid x \in K\})$ is also a k-tree.*

By construction, a k-tree with n nodes has $\binom{k}{2} + k(n-k)$ edges, $n-k$ cliques on $k+1$ nodes, and $k(n-k)+1$ cliques on k nodes. Since every T_k with k or $k+1$ nodes is simply a clique, in the following we will assume $n \geq k+2$.

In a k-tree nodes of degree k are called k-leaves. Note that the neighborhood of each k-leaf forms a clique and then k-leaves are simplicial nodes. A rooted k-tree is a k-tree in which one of its k-cliques $R = \{r_1, r_2, \ldots, r_k\}$ is distinguished; this clique is called the root.

Remark 1. In an unrooted k-tree T_k there are at least two k-leaves; when T_k is rooted at R at least one of those k-leaves does not belong to R (see [21]). Since k-trees are perfect elimination order graphs [22], when a k-leaf is removed from a k-tree the resulting graph is still a k-tree and at most one of its adjacent nodes may became a k-leaf, unless the resulting k-tree is nothing more than a single clique.

In this paper we will deal with k-trees labeled with distinct integer values in $[1, n]$. In Figure 1(a) an example of k-tree with $k = 3$ and 11 nodes labeled with integers in $[1, 11]$ is given. The same k-tree, rooted at $R = \{2, 3, 9\}$, is given in Figure 1(b).

Let us call $\mathcal{T}_k^n$ the set of labeled k-trees with n nodes. It is well known that [1, 12, 16, 21]:

$$|\mathcal{T}_k^n| = \binom{n}{k}(k(n-k)+1)^{n-k-2}$$

When $k = 1$ the set $\mathcal{T}_1^n$ is the set of Cayley's trees and $|\mathcal{T}_1^n| = n^{n-2}$, i.e. the well-known Cayley's theorem.

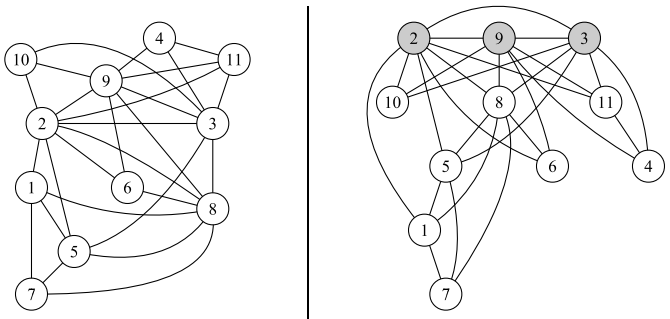

Fig. 1. (a) An unrooted 3-tree T_3 on 11 nodes. (b) T_3 rooted at the clique $\{2, 3, 9\}$.

Definition 2. *[21] A Rényi k-tree R_k is a k-tree with n nodes labeled in $[1, n]$ rooted at the fixed k-clique $R = \{n-k+1, n-k+2, \ldots, n\}$.*

It has been proven [16, 21] that:

$$|\mathcal{R}_k^n| = (k(n-k)+1)^{n-k-1}$$

where $\mathcal{R}_k^n$ is the set of Rényi k-trees with n nodes. It is obvious that $\mathcal{R}_k^n \subseteq \mathcal{T}_k^n$; the equality holds only when $k = 1$ (i.e. the set of labeled trees rooted in n is equivalent to the set of unrooted labeled trees) and when $n = k$ or $n = k+1$ (i.e. the k-tree is a single clique).

3 Characteristic Tree

Here we introduce the *characteristic tree* $T(R_k)$ of a Rényi k-tree that will be used to design our algorithm for coding a generic k-tree.

Let us start by introducing the *skeleton* of a Rényi k-tree. Give a a Rényi k-tree R_k its skeleton $S(R_k)$ is defined according to the definition of k-trees:

1. if R_k is a single k-clique R, $S(R_k)$ is a tree with a single node R;
2. let us consider a k-tree R'_k, its skeleton $S(R'_k)$, and a k-clique K in R'_k. If R_k is the k-tree obtained from R'_k by adding a new node v attached to K, then $S(R_k)$ is obtained by adding to $S(R'_k)$ a new node $X = \{v\} \cup K$ and a new edge (X, Y) where Y is the node of $S(R'_k)$ that contains K, at minimum distance from the root.

$S(R_k)$ is well defined, in particular it is always possible to find a node Y containing K in $S(R'_k)$ because K is a clique in $S(R'_k)$. Moreover Y is unique, indeed it is easy to verify that if two nodes in $S(R'_k)$ contain a value v, their lower common ancestor still contains v. Since it holds for all $v \in K$, there always exists a unique node Y containing K at minimum distance from the root.
The characteristic tree $T(R_k)$ is obtained by labeling nodes and edges of $S(R_k)$ as follows:

1. each node $\{v\} \cup K$ with parent X is labeled v. The node R is labeled 0;
2. each edge from node $\{v\} \cup K$ to its parent $\{v'\} \cup K'$ is labeled with the index of the node in K' (considered as an ordered set) that does not appear in K. When the parent is R the edge is labeled ε.

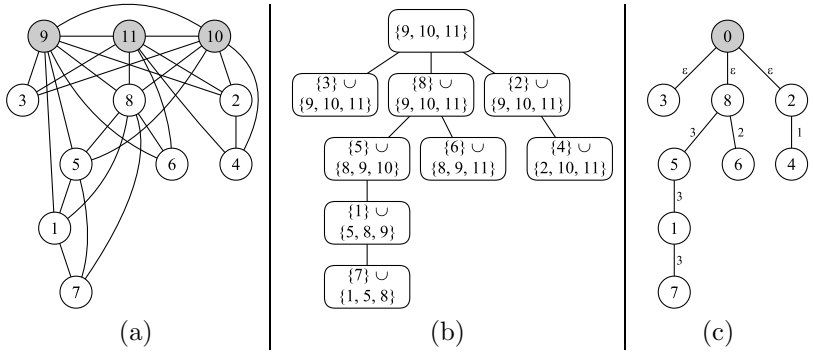

Fig. 2. (a) A Rényi 3-tree R_3 with 11 nodes and root $\{9, 10, 11\}$. (b) The skeleton $S(R_3)$, with nodes $\{v\} \cup K$. (c) The characteristic tree $T(R_3)$.

In Figure 2 a Rényi 3-tree with 11 nodes, its skeleton and its characteristic tree are shown.

It is easy to reconstruct a Rényi k-tree R_k from its characteristic tree $T(R_k)$ since the characteristic tree is well defined and conveys all information needed to rebuild the skeleton of R_k. We point out that there will always be one, and only one, node in K' that does not appear in K (see 2. in the definition of $T(R_k)$). Indeed, v' must appear in K, otherwise $K' = K$ and then the parent of $\{v'\} \cup K'$ would contain K and this would contradict each node in $S(R_k)$ being attached as closely as possible to the root (see 2. in the definition of $S(R_k)$).

Remark 2. For each node $\{v\} \cup K$ of $S(R_k)$ each $w \in K - R$ appears as label of a node in the path from v to 0 in $T(R_k)$.

A linear time algorithm to reconstruct R_k from $T(R_k)$ with a simple traversal of the tree is detailed in Section 6. This algorithm avoids the explicit construction of $S(R_k)$.

Let us consider $\mathcal{Z}_k^n$, the set of all trees with $n-k+1$ nodes labeled with distinct integers in $[0, n-k]$ in which all edges incident on 0 have label ε and all other edges take a label from $[1, k]$. The association between a Rényi k-tree and its characteristic tree is a bijection between $\mathcal{R}_k^n$ and $\mathcal{Z}_k^n$. Obviously, for each Rényi k-tree its characteristic tree belongs to $\mathcal{Z}_k^n$, and this association is invertible. In Section 4 we will show that $|\mathcal{Z}_k^n| = |\mathcal{R}_k^n|$; this will imply the bijectivity of this association.

Our characteristic tree coincides with the *Doubly Labeled Tree* defined in a completely different way in [13] and used in [8]. Our new definition gives us the right perspective to build the tree in linear time, as will be shown in Section 5.

4 Generalized Dandelion code

As stated in the introduction, many codes producing bijection between labeled trees with n nodes and strings of length $n-2$ have been presented in the literature. Here we show a generalization of one of these codes, because we need to take into account labels on edges. We have chosen Dandelion code due to special

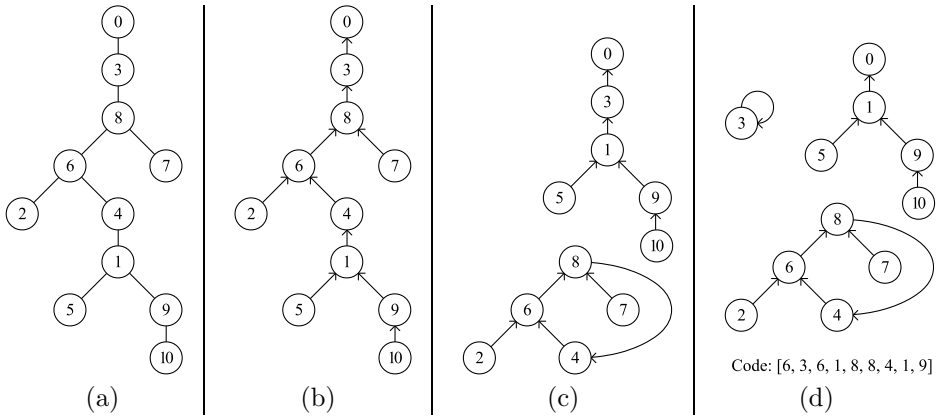

Fig. 3. (a) A simple tree T with 11 nodes labeled in $[0, 10]$. (b) The functional digraph G at the beginning of the Dandelion encoding. (c) G after the first swap $p(1) \leftrightarrow p(8)$. (d) G at the end of the encoding after the swap $p(1) \leftrightarrow p(3)$, together with the code string.

structure of the code strings it produces. This structure will be crucial to ensure the bijectivity at the end of the encoding process of a k-tree (see Section 5 Step 3).

Dandelion code was originally introduced in [10], but its poetic name is due to Picciotto [19]. Our description of this code is based on [5] where linear time coding and decoding algorithms are detailed.

The basic idea behind Dandelion code is to root the tree in 0 and to ensure the existence of edge $(1, 0)$. A tree T with n nodes labeled in $[0, n-1]$ is transformed into a digraph G, such that 0 has no outgoing edges and each node $v \neq 0$ has one outgoing edge; the outgoing edge of 1 is $(1, 0)$. For each v in T, let $p(v)$ be the parent of v in T rooted in 0. G is obtained starting with the edge set $\{(v, p(v)) | v \in V - \{0\}\}$, i.e. initially G is the whole tree with edges oriented upwards to the root (see Figure 3(b)). Now we have to shrink the path between 1 and 0 into a single edge. This can be done by iteratively swapping $p(1)$ with $p(w)$ where $w = max\{u \in path(1, 0)\}$ (see Figure 3(c) and 3(d)). The code string will be the sequence of $p(v)$ for each v from 2 to $n - 1$ in the obtained G.

Since the trees we are dealing with have labels on both nodes and edges, we need to modify the Dandelion code to enable it to hold edge information. In particular, we have to specify what happens when two nodes u and v swap their parents. Our algorithm ensures that the label of the edge $(v, p(v))$ remains associated to $p(v)$. More formally, the new edge $(u, p(v))$ will have the label of the old edge $(v, p(v))$ and similarly the new edge $(v, p(u))$ will have the label of the old edge $(u, p(u))$.

A further natural generalization is to adopt two nodes r and x as parameters instead of the fixed 0 and 1.

In Program 1 the pseudo-code for the generalized Dandelion Code is given; $l(u, v)$ is the label of edge (u, v). The tree T is considered as rooted in r and it is represented by the parent vector p. The condition of Line 2 can be precomputed

for each node in the path between x and r with a simple traversal of the tree, so the linear time complexity of the algorithm is guaranteed.

Program 1 Generalized Dandelion Code

```
1: for v from x to r do
2:   if v = max{w ∈ path(v, r)} then
3:     swap p(x) and p(v), together swap l(x, p(x)) and l(v, p(v))
4: for v ∈ V(T) − {x, r} in increasing order do
5:   append (p(v), l(v, p(v))) to the code
```

The decoding algorithm proceeds to break cycles and loops in the digraph obtained by a given string, and to reconstruct the original path from x to r. We refer to [5] for details.

As an example consider the coding of tree shown in Figure 2(c) with $r = 0$ and $x = 1$, the only swap that occurs is between $p(1)$ and $p(8)$. The code string obtained is: $[(0, \varepsilon), (0, \varepsilon), (2, 1), (8, 3), (8, 2), (1, 3), (5, 3)]$.

As mentioned in the previous section, we now exploit the Generalized Dandelion code to show that $|\mathcal{Z}_k^n| = |\mathcal{R}_k^n|$. Each tree in $\mathcal{Z}_k^n$ has $n - k + 1$ nodes and therefore is represented by a code string of length $n - k - 1$. Each element of this string is either $(0, \varepsilon)$ or a pair in $[1, n - k] \times [1, k]$. Then there are exactly $k(n - k) + 1$ possible pairs. This implies that there are $(k(n - k) + 1)^{n-k-1}$ possible strings, thus proving $|\mathcal{Z}_k^n| = (k(n - k) + 1)^{n-k-1} = |\mathcal{R}_k^n|$.

5 A Linear Time Algorithm for Coding k-trees

In this section we present a new bijective code for k-trees and we show that this code permits linear time encoding and decoding algorithms. To the best of our knowledge, this is the first bijective encoding of k-trees with efficient implementation. In [11] a bijective code for k-trees was presented, but it is very complex and does not seem to permit efficient implementation.

In our algorithm, initially, we have to root the k-tree T_k in a particular clique Q and perform a relabeling to obtain a Rényi k-tree R_k. Then, exploiting the characteristic tree $T(R_k)$ and the Generalized Dandelion code, we bijectively encode R_k. The most demanding step of this process is the computation of $T(R_k)$ starting from R_k and viceversa. This will be shown to require linear time.

Notice that the coding presented in [8], which deals with Rényi k-trees, is not suitable to be extended to obtain a non redundant code for general k-trees.

As noted at the end of the previous section, using the Generalized Dandelion Code, we are able to associate elements in $\mathcal{R}_k^n$ with strings in:

$$\mathcal{B}_k^n = (\{(0, \varepsilon)\} \cup ([1, n - k] \times [1, k]))^{n-k-1}$$

Since we want to encode all k-trees, rather than just Rényi k-trees, our final code will consist of a substring of the Generalized Dandelion Code for $T(R_k)$, together with information concerning the relabeling used to transform T_k into R_k.

Codes for k-trees associate elements in $\mathcal{T}_k^n$ with elements in:

$$\mathcal{A}_k^n = \binom{[1,n]}{k} \times (\{(0, \varepsilon)\} \cup ([1, n - k] \times [1, k]))^{n-k-2}$$

The obtained code is bijective: this will be proved by a decoding process that is able to associate to each code in $\mathcal{A}_{n,k}$ its corresponding k-tree. Note that $|\mathcal{A}_k^n| = |\mathcal{T}_k^n|$.

The encoding algorithm is summarized in the following 4 steps:

Coding Algorithm
Input: a k-tree T_k
Output: a code in $\mathcal{A}_{n,k}$

1. Identify Q, the k-clique adjacent to the maximum labeled leaf l_M of T_k. By a relabeling process ϕ, transform T_k into a Rényi k-tree R_k.
2. Generate the characteristic tree T for R_k.
3. Compute the generalized Dandelion Code for T using as parameters $r = 0$ and $x = \phi(\overline{q})$, where $\overline{q} = \min\{v \notin Q\}$. Remove from the obtained code string S the pair corresponding to $\phi(l_M)$.
4. Return the code (Q, S).

Assuming that the input k-tree is represented by adjacency lists adj, we detail how to implement the first three steps of our algorithm in linear time.

Step 1 Compute the degree $d(v)$ of each node v and find l_M, i.e. the maximum v such that $d(v) = k$, then the node set Q is $adj(l_M)$. In order to obtain a Rényi k-tree, nodes in Q have to be associated with values in $\{n-k+1, n-k+2, \ldots, n\}$. This relabeling can be described by a permutation ϕ defined by the following three rules:

1. if q_i is the i-th smallest node in Q, assign $\phi(q_i) = n - k + i$;
2. for each $q \notin Q \cup \{n-k+1, \ldots, n\}$, assign $\phi(q) = q$;
3. unassigned values are used to close permutation cycles, formally: for each $q \in \{n-k+1, \ldots, n\} - Q$, $\phi(q) = i$ such that $\phi^j(i) = q$ and j is maximized.

Figure 4 provides a graphical representation of the permutation ϕ corresponding to the 3-tree in Figure 1(a), where $Q = \{2, 3, 9\}$, obtained as the neighborhood of $l_M = 10$. Forward arrows correspond to values assigned by rule 1, small loops are those derived from rule 2, while backward arrows closing cycles are due to rule 3.

Fig. 4. Graphical representation of ϕ for 3-tree in Figure 1(a).

The Rényi k-tree R_k is T_k relabeled by ϕ. The final operation of this step is to order the adjacency lists of R_k. The reason for this will be clear in the next step.

Figure 2(a) gives the Rényi 3-tree R_3 obtained by relabeling the T_3 of Figure 1(a) by ϕ represented in Figure 4. The root of R_3 is $\{9, 10, 11\}$.

Let us now prove that the overall time complexity of step 1 is $O(nk)$. The computation of $d(v)$ for each node v can be implemented by scanning all adjacency lists of T_k. Since a k-tree with n nodes has $\binom{k}{2} + k(n-k)$ edges, it requires $O(nk)$ time, which is linear with respect to the input size.

The procedure to compute ϕ in $O(n)$ time is given in Program 2:

Program 2 Compute ϕ

```
1: for q_i ∈ Q in increasing order do
2:    φ(q_i) = n − k + i
3: for i = 1 to n − k do
4:    j = i
5:    while φ(j) is assigned do
6:       j = φ(j)
7:    φ(j) = i
```

Assignments of rule 1 are made by the loop in Line 1, in which it is assumed that elements in Q appear in increasing order. The loop in Line 3 implements rules 2 and 3 in linear time. Indeed the while loop condition of Line 5 is always false for all those values not belonging to $Q \cup \{n-k+1, \ldots, n\}$. For remaining values the inner while loop scans each permutation cycle only once, according to rule 3 of the definition of ϕ.

Relabeling all nodes of T_k to obtain R_k requires $O(nk)$ operations, as well as the procedure in Program 3 used to order its adjacency lists.

Program 3 Order Adjacency Lists

```
1: for i = 1 to n do
2:    for each j ∈ adj(i) do
3:       append i to newadj(j)
4: return newadj
```

Step 2 The goal of this step is to build the characteristic tree T of R_k. In order to guarantee linear time complexity we avoid the explicit construction of the skeleton $S(R_k)$. We build the node set and the edge set of T separately.

The node set is computed identifying all maximal cliques in R_k; this can be done by pruning R_k from k-leaves. The pruning process proceeds by scanning the adjacency lists in increasing order: whenever it finds a node v with degree k, a node in T labeled by v, representing the maximal clique with node set $v \cup adj(v)$, is created. Then v is removed from R_k and consequently the degree of each of its adjacent nodes is decreased by one.

In a real implementation of the pruning process, in order to limit time complexity, the explicit removal of each node should be avoided, keeping this information by marking removed nodes and decreasing node degrees. When v becomes a k-leaf, the node set identifying its maximal clique is given by v union the nodes in the adjacent list of v that have not yet been marked as removed. We will store this subset of the adjacency list of v as K_v, it is a list of exactly k integers.

Note that, when v is removed, at most one of its adjacent nodes becomes a k-leaf (see Remark 1). If this happens, the pruning process selects the minimum between the new k-leaf and the next k-leaf in the adjacency list scan.

At the end of this process the original Rényi k-tree is reduced to its root $R = \{n-k+1, \ldots, n\}$. To represent this k-clique the node labeled 0 is added to T (the algorithm also assigns $K_0 = R$).

This procedure is detailed in Program 4; its overall time complexity is $O(nk)$. Indeed, it removes $n-k$ nodes and each removal requires $O(k)$ time.

Program 4 Prune R_k

function remove(x)

```
1: let K_x be adj(x) without all marked elements
2: create a new node in T with label x        // it corresponds to node
   {x} ∪ K_x of the skeleton
3: mark x as removed
4: for each unmarked y ∈ adj(x) do
5:    d(y) = d(y) − 1
```

main

```
1: for v = 1 to n − k do
2:    w = v
3:    if d(w) = k then
4:       remove(w)
5:       while ∃ an unmarked u ∈ adj(w) s.t u < v and d(u) = k do
6:          w = u
7:          remove(w)
```

In order to build the edge set, let us consider for each node v the set of its eligible parents, i.e. all w in K_v. Since all eligible parents must occur in the ascending path from v to the root 0 (see Remark 2), the correct parent is the one at maximum distance from the root. This is the reason why we proceed following the reversed pruning order.

The edge set is represented by a vector p identifying the parent of each node. 0 is the parent of all those nodes s.t. $K_v = R$. The level of these nodes is 1.

To keep track of the pruning order, nodes can be pushed into a stack during the pruning process. Now, following the reversed pruning order, as soon as a node v is popped from the stack, it is attached to the node in K_v at maximum level. We assume that the level of nodes in R (which do not belong to T) is 0.

The pseudo-code of this part of Step 2 is shown in Program 5.

Program 5 Add edges

```
1: for each v ∈ [1, n − k] in reversed pruning order do
2:    if K_v = R then
3:       p(v) = 0
4:       level(v) = 1
5:    else
6:       choose w ∈ K_v s.t. level(w) is maximum
7:       p(v) = w
8:       level(v) = level(w) + 1
```

The algorithm of Program 5 requires $O(nk)$ time. In fact, it assigns the parent of $n-k$ nodes, each assignment involves the computation of the maximum (Line 6) and requires k comparisons.

To complete step 2 it only remains to label each edge $(v, p(v))$. When $p(v) = 0$, the label is ε; in general, the label $l(v, p(v))$ must receive the index of the only element in $K_{p(v)}$ that does not belong to K_v. This information can be computed in $O(nk)$ by simple scans of lists K_v. The ordering of the whole adjacency list made at the end of step 1 ensures that elements in all K_v appear in increasing order.

Figure 2(c) shows the characteristic tree computed for the Rényi 3-tree of Figure 2(a).

Step 3 Applying the generalized Dandelion Code with parameters 0 and $x = \phi(\overline{q})$, where $\overline{q} = \min\{v \notin Q\}$, we obtain a code S consisting in a list of $n-k-1$ pairs. For each $v \in \{1, 2, \ldots, n-k\} - \{x\}$ there is a pair $(p(v), l(v, p(v)))$ taken from the set $\{(0, \varepsilon)\} \cup ([1, n-k] \times [1, k])$. Given all this, the obtained code is redundant because we already know, from the relabeling process performed in Step 1, that the greatest leaf l_M of T_k corresponds to a child of the root in T. Therefore the pair associated to $\phi(l_M)$ must be $(0, \varepsilon)$ and can be omitted. The Generalized Dandelion code already omits the information $(0, \varepsilon)$ associated with the node x, so, in order to reduce the code length, we need to guarantee that $\phi(l_M) \neq x$. We already know that a k-tree on $n \geq k+2$ nodes has at least 2 k-leaves. As Q is chosen as the adjacent k-clique of the maximum leaf l_M it cannot contain a k-leaf. So there exists at least a k-leaf less than l_M that does not belong to Q; $\overline{q}$ will be less or equal to this k-leaf. Consequently $\overline{q} \neq l_M$ and, since ϕ is a permutation, $\phi(l_M) \neq \phi(\overline{q})$. The removal of the redundant pair from the code S completes Step 3.

Since the Generalized Dandelion Code can be computed in linear time, the overall time complexity of the coding algorithm is $O(nk)$.

We want to remark that we choose Dandelion Code because it allows us to easily identify an information (the pair $(0, \varepsilon)$ associated to $\phi(l_M)$) that can be removed in order to reduce the code length from $n-k-1$ to $n-k-2$: this is crucial to obtain a bijective code for all k-trees.

Many other known codes for Cayley's trees can be generalized to encode edge labeled trees, obtaining bijection between Rényi k-trees and strings in $\mathcal{B}_{n,k}$. However other known codes, such as all Prüfer-like codes, do not make it possible to identify a removable redundant pair. This means that not any code for Rényi k-trees can be exploited to obtain a code for k-trees.

The returned pair (Q, S) belongs to $\mathcal{A}_{n,k}$, since $Q \in \binom{[1,n]}{k}$, and S is a string of pairs obtained by removing a pair from a string in $\mathcal{B}_{n,k}$. Due to lack of space we cannot discuss here how this pair can be efficiently represented in $\lceil \log_2(|\mathcal{A}_{n,k}|) \rceil$ bits.

The Dandelion Code obtained from the characteristic tree in Figure 2(c) with parameters $r = 0$ and $x = 1$ is: $[(0, \varepsilon), (0, \varepsilon), (2, 1), (8, 3), (8, 2), (1, 3), (5, 3)] \in \mathcal{B}_3^{11}$; this is a code for the Rényi 3-tree in Figure 2(a). The 3-tree T_3 in Figure 1(a) is coded as: $(\{2, 3, 9\}, [(0, \varepsilon), (2, 1), (8, 3), (8, 2), (1, 3), (5, 3)]) \in \mathcal{A}_3^{11}$. We recall that in this example $Q = \{2, 3, 9\}$, $l_M = 10$, $\overline{q} = 1$, $\phi(l_M) = 3$, and $\phi(\overline{q}) = 1$.

6 A Linear Time Algorithm for Decoding k-trees

Any pair $(Q, S) \in \mathcal{A}_{n,k}$ can be decoded to obtain a k-tree whose code is (Q, S). This process can be performed with the following algorithm:

Decoding Algorithm
Input: a code (Q, S) in $\mathcal{A}_{n,k}$
Output: a k-tree T_k

1. Compute ϕ starting from Q and find l_M and $\overline{q}$.
2. Insert the pair $(0, \varepsilon)$ corresponding to $\phi(l_M)$ in S and decode it to obtain T.
3. Rebuild the Rényi k-tree R_k by visiting T.
4. Apply ϕ^{-1} to R_k to obtain T_k.

Let us detail the decoding algorithm. Once Q is known, it is possible to compute $\overline{q} = \min\{v \in [1, n] | v \notin Q\}$ and ϕ as described in Step 1 of coding algorithm. Since all internal nodes of T explicitly appear in S, it is easy to derive set L of all leaves of T by a simple scan of S. Note that leaves in T coincide with k-leaves in R_k. Applying ϕ^{-1} to all elements in L we can reconstruct the set of all k-leaves of the original T_k, and therefore find l_M, the maximum leaf in T_k.

In order to decode S, a pair $(0, \varepsilon)$ corresponding to $\phi(l_M)$ needs to be added, and then the decoding phase of the Generalized Dandelion Code with parameters $\phi(\overline{q})$ and 0 applied. The obtained tree T is represented by its parent vector.

Program 6 Rebuild R_k

```
1: initialize R_k as the k-clique R on {n-k+1, n-k+2, ..., n}
2: for each v in T in breadth first order do
3:   if p(v) = 0 then
4:     K_v = R in increasing order
5:   else
6:     let w be the element of index l(v, p(v)) in K_{p(v)}
7:     K_v = K_{p(v)} - {w} ∪ {p(v)} in increasing order
8:   add v to R_k
9:   add to R_k all edges (u, v) s.t. u ∈ K_v
```

The reconstruction of the Rényi k-tree R_k is detailed in Program 6. Finally, T_k is obtained by applying ϕ^{-1} to R_k.

The overall complexity of the decoding algorithm is $O(nk)$. In fact the only step of the algorithm that requires some explanation is Line 7 of Program 6. Assuming that $K_{p(v)}$ is ordered, to create K_v in increasing order, $K_p(v)$ simply needs to be scanned omitting w and inserting $p(v)$ in the correct position. As $K_0 = \{n - k + 1, \ldots, n\}$ is trivially ordered, all K_v will be ordered.

7 Conclusions

In this paper we have introduced a new bijective code for labeled k-trees which, to the best of our knowledge, produces the first encoding and decoding algorithms running in linear time with respect to the size of the k-tree.

In order to develop our bijective code for k-trees we passed through a transformation of a k-tree in a Rényi k-tree and developed a new coding for Rényi

k-trees based on a generalization of the Dandelion code. The choose of Dandelion code is motivated by the need of identifying and discarding a redundant information. This is crucial to ensure the resulting code for k-trees to be bijective.

All details needed to obtain linear time implementations for encoding and decoding algorithms have been presented.

References

1. L.W. Beineke and R.E. Pippert. On the Number of k-Dimensional Trees. *Journal of Combinatorial Theory*, 6:200–205, 1969.
2. H.L. Bodlaender. A Tourist Guide Through Treewidth. *Acta Cybernetica*, 11:1–21, 1993.
3. H.L. Bodlaender. A Partial k-Arboretum of Graphs with Bounded Treewidth. *Theoretical Computer Science*, 209:1–45, 1998.
4. S. Caminiti, I. Finocchi, and R. Petreschi. A Unified Approach to Coding Labeled Trees. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN '04), LNCS 2976*, pages 339–348, 2004.
5. S. Caminiti and R. Petreschi. String Coding of Trees with Locality and Heritability. In *Proceedings of the 11th International Conference on Computing and Combinatorics (COCOON '05), LNCS 3595*, pages 251–262, 2005.
6. A. Cayley. A Theorem on Trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.
7. W.Y.C. Chen. A general bijective algorithm for trees. *Proceedings of the National Academy of Science, USA*, 87:9635–9639, 1990.
8. W.Y.C. Chen. A Coding Algorithm for Rényi Trees. *Journal of Combinatorial Theory*, 63A:11–25, 1993.
9. N. Deo and P. Micikevičius. A New Encoding for Labeled Trees Employing a Stack and a Queue. *Bulletin of the Institute of Combinatorics and its Applications (ICA)*, 34:77–85, 2002.
10. Ö. Eğecioğlu and J.B. Remmel. Bijections for Cayley Trees, Spanning Trees, and Their q-Analogues. *Journal of Combinatorial Theory*, 42A(1):15–30, 1986.
11. Ö. Eğecioğlu and L.P. Shen. A Bijective Proof for the Number of Labeled q-Trees. *Ars Combinatoria*, 25B:3–30, 1988.
12. D. Foata. Enumerating k-Trees. *Discrete Mathematics*, 1(2):181–186, 1971.
13. C. Greene and G.A. Iba. Cayley's Formula for Multidimensional Trees. *Discrete Mathematics*, 13:1–11, 1975.
14. F. Harary and E.M. Palmer. on Acyclic Simplicial Complexes. *Mathematika*, 15:115–122, 1968.
15. L. Markenzon, P.R. Costa Pereira, and O. Vernet. The Reduced Prüfer Code for Rooted Labelled k-Trees. In *Proceedings of 7th International Colloquium on Graph Theory, Electronic Notes in Discrete Mathematics*, volume 22, pages 135–139, 2005.
16. J.W. Moon. The Number of Labeled $k-$Trees. *Journal of Combinatorial Theory*, 6:196–199, 1969.
17. J.W. Moon. *Counting Labeled Trees.* William Clowes and Sons, London, 1970.
18. E.H. Neville. The Codifying of Tree-Structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.
19. S. Picciotto. *How to Encode a Tree.* PhD thesis, University of California, San Diego, 1999.
20. H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:142–144, 1918.
21. A. Rényi and C. Rényi. The Prüfer Code for k-Trees. In P. Erdös *at al.*, editor, *Combinatorial Theory and its Applications*, pages 945–971, North-Holland, Amsterdam, 1970.
22. D.J. Rose. On Simple Characterizations of k-Trees. *Discrete Mathematics*, 7:317–322, 1974.