Bijective Linear Time Coding and Decoding for k-Trees ⋆

Saverio Caminiti, Emanuele G. Fusco, and Rossella Petreschi

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 of graphs, 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, the first bijective code for trees, 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 coding 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 [8], has been widely studied in the literature. Coding labeled trees by means of strings of node 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 [14], random trees generation [12], data compression [11], and computation of forest volumes of graphs [20]). Several different bijective codes that realize associations between labeled trees and strings of labels have been introduced, see for example [9, 13, 15, 23–26]. From an algorithmic point of view, the problem has been investigated thoroughly and optimal coding and decoding algorithms for these codes are known [4, 7, 9, 13].

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

⋆ Work partially supported by MIUR: Italian Ministry for University and Scientific Research. A preliminary version of this paper appeared in the Proceedings of the IntErnational Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07) [6].

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 [27] 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 [10, 16], together with coding and decoding algorithms. In [21], the authors presented linear time algorithms for coding and decoding Rényi k-trees by means of the redundant Prüfer code. This code is not bijective and the decoding algorithm proposed fails on strings that are not valid codewords. To the best of our knowledge, this paper is the first one that explicitly provides efficient algorithms to bijectively code and decode k-trees. Moreover our code is suitable for Rényi k-trees and arbitrarily rooted k-trees as well.

The paper is organized as follows: in Section 2 we provide a background on k-trees. In Section 3 and 4 we introduce two building blocks of our coding technique (characteristic tree and Generalized Dandelion Code) while all details for the coding and decoding procedures are given in Section 5 and 6. In Section 7 we discuss the physical representation of our code. The paper ends with some conclusions and future directions for research in this topic.

2 Preliminaries

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

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

Definition 1. *[19] A k-tree is defined in the following recursive way:*

1. *A k-clique 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 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 k-tree 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 with a distinguished k-clique $R = \{r_1, r_2, \ldots, r_k\}$; R is called the root of the rooted k-tree.

Remark 1. Each k-tree T_k with $n \geq k+2$ nodes contains at least two k-leaves; when T_k is rooted at R at least one of those k-leaves does not belong to R

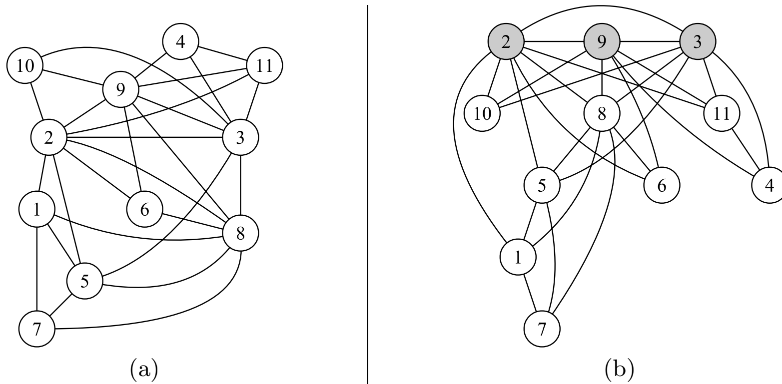

Fig. 1. (a) An unrooted 3-tree T_3 with 11 nodes. (b) T_3 rooted at the clique $\{2, 3, 9\}$.

(see [27]). Since k-trees are perfect elimination order graphs [28], when a k-leaf is removed from a k-tree the resulting graph is still a k-tree. If the resulting k-tree is not a clique, at most one node adjacent to the removed k-leaf may became a k-leaf.

In Figure 1(a) we give an example of a k-tree with $k = 3$ and 11 nodes labeled with integers in $[1, 11]$. 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 k-trees with n nodes labeled with distinct labels. The cardinality of $\mathcal{T}_k^n$ is (see [1, 17, 22, 27]):

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

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.

Arbitrarily rooted k-trees with n nodes labeled with distinct labels can be denoted as a pair $(\mathcal{T}_k^n, R)$. Since each k-tree T_k contains $k(n-k)+1$ cliques on k nodes, the number of arbitrarily rooted k-trees is:

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

Without loss of generality, in the rest of this paper we will use integers in $[1, n]$ as labels for a k-tree of n nodes.

Definition 2. *[27] A Rényi k-tree R_k is a rooted k-tree with n nodes labeled in $[1, n]$ and root $R = \{n-k+1, n-k+2, \ldots, n\}$.*

It has been proven [22, 27] 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.

Remark 2. Note that the set of labeled trees rooted at n is equivalent to the set of unrooted labeled trees. This equivalence cannot be transposed on k-trees when $k > 1$. Indeed, not all k-trees contain the clique $\{n-k+1, n-k+2, \ldots, n\}$ and then not all k-trees are eligible to be Rényi k-trees. This implies $\mathcal{R}_k^n \subseteq \mathcal{T}_k^n$.

3 Characteristic Tree

In this section we introduce the *characteristic tree* of a rooted k-tree. This structure is fundamental in the rest of this paper as we will use the characteristic tree of a Rényi k-tree in our coding process.

Let us start by introducing the *skeleton* of a rooted k-tree.

Definition 3. *Given a rooted k-tree T_k with root R, obtainable by T'_k rooted in R by adding a new node v connected to a k-clique K (see Definition 1), the skeleton $S(T_k, R)$ is obtained by adding to $S(T'_k, R)$ a new node $X = \{v\} \cup K$ and a new edge (X, Y). Y is the node of $S(T'_k, R)$ that contains K at minimum distance from the root. If T_k is the single k-clique R, its skeleton $S(T_k, R)$ is a tree with a single node R.*

The skeleton $S(T_k, R)$ of a rooted k-tree T_k with root R is well defined: indeed it is always possible to find a node Y containing K in T'_k because K is a clique in $S(T'_k, R)$. Moreover Y is unique: it is easy to verify that if two nodes in $S(T'_k, R)$ contain a value v, their lowest common ancestor still contains v. Since this holds for all $v \in K$, there always exists a unique node Y containing K at minimum distance from the root.

Definition 4. *The characteristic tree $T(T_k, R)$ of a rooted k-tree T_k with root R is obtained by labeling nodes and edges of $S(T_k, R)$ as follows:*

1. *Node R is labeled* 0 *and each node $\{v\} \cup K$ is labeled v;*
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.*

The existence of a unique node in $K' \setminus K$ is guaranteed by Definition 3. Indeed, v' must appear in K, otherwise $K' = K$ and the parent of $\{v'\} \cup K'$ contains K. This contradicts the fact that each node in $S(T_k, R)$ is attached at minimum distance from the root.

Remark 3. For each node $\{v\} \cup K$ of $S(T_k, R)$, each $w \in K \setminus R$ appears as label of a node in the path from v to 0 in $T(T_k, R)$.

As we mentioned before, in our code we will use the characteristic tree of a Rényi k-trees R_k. As in Rényi k-trees the root is fixed, we omit the argument R, referring the skeleton as $S(R_k)$ and the characteristic tree as $T(R_k)$.

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

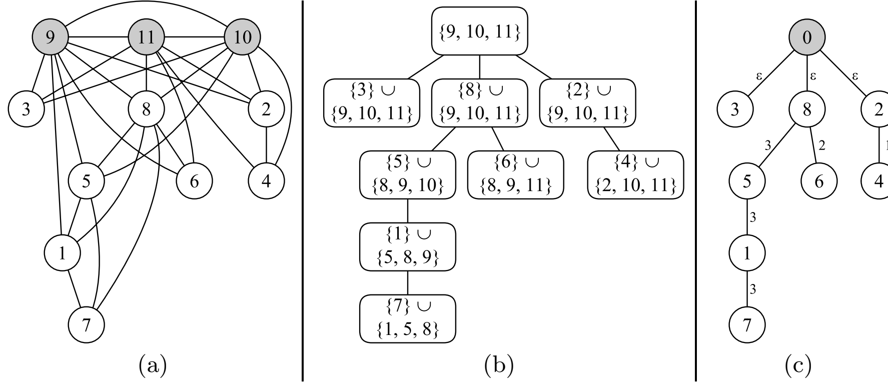

Fig. 2. (a) A Rényi 3-tree R_3 with 11 nodes and root $\{9, 10, 11\}$. (b) The skeleton of R_3, with nodes $\{v\} \cup K$. (c) The characteristic tree of R_3.

It is easy to see that, given a characteristic tree T, it is possible to reconstruct the corresponding Rényi k-tree: indeed the reconstruction of the skeleton from T is straightforward, and the skeleton tell us, for each node, which clique the node should be connected to.

We are interested in finding algorithms to compute $T(R_k)$ from R_k and vice versa in linear time. In order to satisfy this constraint the algorithms detailed in the following sections will avoid the explicit construction of the skeleton. Moreover, we have to remark that, when restricted to Rényi k-trees, our characteristic tree coincides with the *Doubly Labeled Tree* defined in a completely different way in [18] and used in [10]. Our new definition gives us the right perspective to obtain linear time algorithms.

At the end of this section, let us consider the class of all characteristic trees of Rényi k-trees: $\mathcal{Z}_k^n$. More formally, $\mathcal{Z}_k^n$ is 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 $\mathcal{Z}_k^n$. Indeed, 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 $|\mathcal{Z}_k^n| = |\mathcal{R}_k^n|$, proving that the association between a Rényi k-tree and its characteristic tree is a bijection.

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, that takes into account labels on edges and can be use to code characteristic trees of Rényi k-trees. We have chosen Dandelion code due to the special structure of the code strings it produces. This structure will be crucial to ensure the bijectivity at the end of the coding process of a k-tree (see Section 5 Step 3).

Dandelion code was originally introduced in [15], but its poetic name is due to Picciotto [25]. Our description of this code is based on [7] where linear time coding and decoding algorithms are detailed.

The basic idea behind Dandelion code is to root the tree at 0 and to transform it to ensure the existence of edge $(1,0)$. Performing these operations, the parent vector of the transformed tree will contain two useless information $p(0)$ and $p(1)$, whose elimination leads to a $n-2$ labels representation of the tree.

The Generalized Dandelion Code takes as parameters r and x. It considers a tree T, with n nodes with distinct labels in $[0, n-1]$, and an edge labeling function ℓ such that: each edge incident on r has label ε and all other edges have label over a given alphabet Σ. At the beginning of the coding procedure T is rooted at r, thus identifying, for each node v, its parent $p(v)$. Considering T as a digraph with edges oriented upwards, T is the functional digraph of the function p. The existence of the oriented edge (x, r) is guaranteed by breaking the original path between x and r into cycles (or loops): until $p(x) \neq r$, $p(x)$ is swapped with $p(w)$, where w is the node with maximum label in the ascending path from x to r.

At each swap a new cycle is introduced: node w is connected either to itself in a loop or with a node belonging to its subtree. As each parent swap changes the set of graph edges, we should specify what happens with edge labels. Our algorithm ensures that the edge labels remain associated to parent nodes. More formally, the new edge $(x, p(w))$ will have the label of the old edge $(w, p(w))$ and the new edge $(w, p(x))$ will have the label of the old edge $(x, p(x))$.

The graph resulting from the coding process satisfies the following invariants:

- node r has no outgoing edges,
- each node except r has exactly one outgoing edge,
- the outgoing edge of node x is (x, r),
- each edge incoming in r has label ε.

Exploiting the invariants, the resulting graph can be univocally represented by $p(v)$ and $\ell(v, p(v))$ for each $v \in [0, n-1] \setminus \{r, x\}$. The sequence of these $n-2$ pairs constitutes the Generalized Dandelion Code of the original tree T.

In Program 1 the coding algorithm is detailed.

Program 1 Generalized Dandelion Coding

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

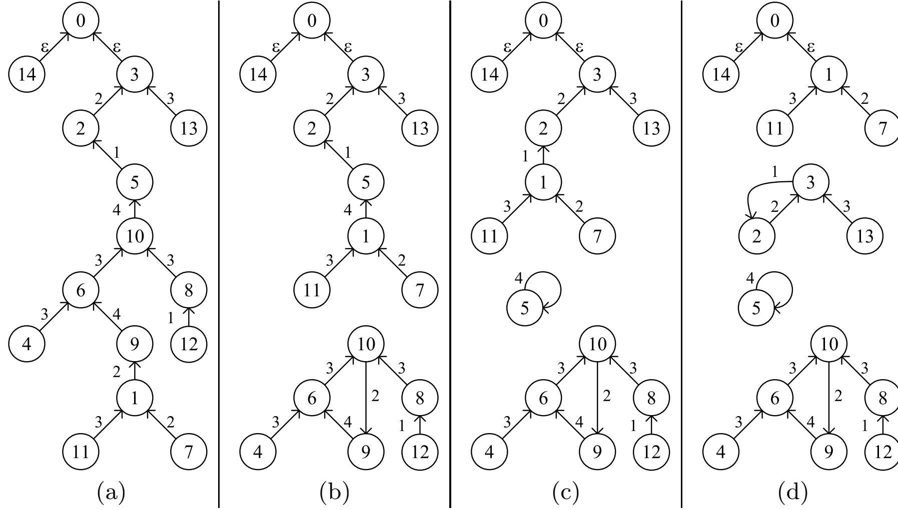

Fig. 3. (a) A tree T with 15 nodes labeled in $[0, 14]$ and edge labels in $[1, 4]$, represented as rooted at 0. (b) After the first swap $p(1) \leftrightarrow p(10)$, cycle $\{10, 9, 6\}$ has been introduced. (c) a loop 5 has been introduced, after the second swap $p(1) \leftrightarrow p(5)$. (d) The tree T at the end of the coding, after the last swap $p(1) \leftrightarrow p(3)$. The codeword is $[(3, 2), (2, 1), (6, 3), (5, 4), (10, 3), (1, 2), (10, 3), (6, 4), (9, 2), (1, 3), (8, 1), (3, 3), (0, \varepsilon)]$.

The condition of Line 2 can be precomputed for each node with a simple traversal of the path between x and r, so the linear time complexity of the algorithm is guaranteed.

In Figure 3 and example of Generalized Dandelion Coding, with parameters $r = 0$ and $x = 1$, is presented.

As a further example let us code (with $r = 0$ and $x = 1$) the tree shown in Figure 2(c). Here the only swap occurring is $p(1) \leftrightarrow p(8)$. The code string obtained is: $[(0, \varepsilon), (0, \varepsilon), (2, 1), (8, 3), (8, 2), (1, 3), (5, 3)]$.

Remark 4. During the coding, the set of parents $\{p(v) : v \in T\}$ does not change because we never introduce new parents or remove them, only swaps may occur. Since the codeword is generated using node parents, each internal node of the original tree appears in the codeword (except perhaps the root r) and vice versa each first element of a pair in the codeword is an internal node in the original tree.

Let us now detail how it is possible to reconstruct the tree from its codeword S. S is a sequence of $n-2$ pairs, each pair is either (r, ε) or a pair in $([0, n-1] \setminus \{r\}) \times \Sigma$.

Initially we construct a functional digraph G, whose node set is $[0, n-1]$, in the following way: consider all nodes except r and x, in increasing order. Let v_i be the i-th node and let (p_i, l_i) be the i-th pair in S. Add to G the oriented

edges (v_i, p_i) with label l_i, for each v_i, and the oriented edge (x, r) with label (r, ε).

The decoding proceeds breaking all cycles (or loops) in G in order to transform G into a tree by correctly reconstructing the path between x and r. Let $\{C_1, C_2, \ldots, C_j\}$ be the set of cycles in G and let m_i be the node of maximum label in C_i: each m_i is a node whose parent has been swapped with $p(x)$ in the coding. So it is sufficient to swap back $p(x)$ and $p(m_i)$, considering all m_i from the smallest to the largest. Edge labels remain associated to parent nodes.

In the decoding algorithm detailed in Program 2, the graph G is represented with a vector p keeping, for each node, the endpoint of its outgoing edge (analogous to parent vector for trees). This can be done because the outdegree of nodes in G is at most 1.

Program 2 Generalized Dandelion Decoding

```
1. Construct G from S
2. Identify all cycles in G and their maximal nodes
3. for each maximal node m_i in increasing order do
4.    swap ℓ(x,p(x)) and ℓ(m_i,p(m_i))
5.    swap p(x) and p(m_i)
```

Program 2 has linear time complexity. Indeed, Line 2 can be implemented by calling, for each vertex v, the function `analyze` detailed in Program 3. Each edge $(v, p(v))$ is traversed exactly once by function `analyze`; when the function identifies a cycle (Line 3) it calls an auxiliary function that compute the maximal node with a further visit of the cycle.

Program 3 Identify Cycles

```
function analyze(v)
1. if status(v) ≠ processed then
2.    status(v) = inProgress
3.    if status(p(v)) = inProgress then a cycle has been identified
4.    else analyze(p(v))
5.    status(v) = processed
```

As mentioned at the end of 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. The number of possible strings is $(k(n-k)+1)^{n-k-1}$, and then $|\mathcal{Z}_k^n| = (k(n-k)+1)^{n-k-1} = |\mathcal{R}_k^n|$.

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 allows linear time coding and decoding algorithms. To the best of our knowledge, this paper is the first one that explicitly provides efficient algorithms to bijectively code and decode k-trees. In [16] a bijective code for k-trees was presented, but it does not seem to allow efficient implementation.

Our algorithm initially transforms a k-tree in a Rényi k-tree: we root the k-tree T_k at a particular clique Q and we perform a relabeling to obtain a Rényi k-tree R_k. Exploiting the characteristic tree $T(R_k)$ and the Generalized Dandelion Code, we bijectively code R_k. The most demanding step of this process is the computation of $T(R_k)$ starting from R_k. We will show that even this step can be done in linear time.

Notice that the coding presented in [10], 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 code all k-trees, rather than just Rényi k-trees, our final code will consist of a substring of length $n-k-2$ of the Generalized Dandelion Code for $T(R_k)$, together with information describing 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 $\mathcal{A}_{n,k}$ the corresponding k-tree. Note that $|\mathcal{A}_k^n| = |\mathcal{T}_k^n|$.

The coding algorithm is summarized in the following 4 steps:

CODING ALGORITHM
Input: a k-tree T_k with n nodes
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 with $r = 0$ 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 in the following way:

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 Step 1 consists in ordering the adjacency lists of R_k. The reason for this operation 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 4.

Program 4 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
```

Let us show the correspondence between rules in the definition of the function ϕ and lines of Program 4: 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 all other nodes, 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 5 used to order its adjacency lists.

Program 5 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 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 adjacency list of v that have not been marked as removed yet. We will store this subset of the adjacency list of v as K_v: 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 6, its overall time complexity is $O(nk)$. Indeed, it removes $n-k$ nodes and each removal requires $O(k)$ time.

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 root 0 (see Remark 3), the correct parent is the one at maximum distance from the root; so 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.

Program 6 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
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) such that u < v and d(u) = k do
6.          w = u
7.          remove(w)
```

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 7.

Program 7 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 7 requires $O(nk)$ time. In fact, it assigns the parent of $n-k$ nodes, each assignment involves the computation of a maximum (Line 6) and requires k comparisons.

To complete Step 2, it 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. All labels can be computed in $O(nk)$ time by scanning lists K_v, as Program 5 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 $r = 0$ 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 $v \in \{1, 2, \ldots, n-k\} \setminus \{x\}$ there is a pair $(p(v), l(v, p(v)))$ taken from the set $\{(0, \varepsilon)\} \cup ([1, n-k] \times [1, k])$. As it is, the obtained code is redundant because we already know, from the relabeling process performed in Step 1, that the largest 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 $\phi(l_M) \neq x$.

Lemma 1. *Given a k-tree T_k with n nodes, let l_M be the maximum leaf of T_k and ϕ the permutation computed by Program 4. Then, if x is chosen as $\phi(\min\{v \notin Q\})$, it holds $\phi(l_M) \neq x$.*

Proof. From Remark 1, we already know that a k-tree on $n \geq k+2$ nodes has at least 2 k-leaves. Q cannot contain a k-leaf, since it is chosen as the adjacent k-clique of the maximum leaf l_M. So there exists at least a k-leaf smaller than l_M that does not belong to Q; $\overline{q} = \min\{v \notin Q\}$ will be less than or equal to this k-leaf. Consequently $l_M \neq \overline{q}$ 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 $O(nk)$.

It is now clear that we have chosen 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. Indeed, many other known codes for Cayley's trees, such as Prüfer-like codes [5], can be generalized to code edge labeled trees, obtaining bijection between Rényi k-trees and strings in $\mathcal{B}_{n,k}$. However these 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 obtained by removing a pair from a string in $\mathcal{B}_{n,k}$.

The Generalized Dandelion Code obtained from the characteristic tree in Figure 2(c), using as 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 (Program 4). Since all internal nodes of T explicitly appear in S (see Remark 4), it is easy to derive the 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 0 and $\phi(\overline{q})$ has to be applied. The obtained tree T is represented by its parent vector.

The reconstruction of the Rényi k-tree R_k is detailed in Program 8. We assume that each K_v is a list of k integers, in increasing order.

Program 8 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
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)}
8.   add v to R_k
9.   add to R_k all edges (u, v) s.t. u ∈ K_v
```

The last step of the decoding process consists in applying ϕ^{-1} to R_k in order to obtain T_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 8. Assuming that $K_{p(v)}$ is ordered, to create K_v in increasing order, it is enough to scan $K_p(v)$ omitting w and inserting $p(v)$ in the correct position. Since all $K_v = R = \{n - k + 1, \ldots, n\}$ are trivially ordered, our assumption is always verified.

7 Compact Representation

In order to consider every aspect of the problem of efficiently code and decode k-trees, we decide to conclude the paper with some considerations about the physical representation of codewords in computer memories.

The main motivation for this section comes from applications like Random k-tree Generation and Evolutionary Algorithms [14]. In these context codewords are generated and manipulated directly by means of operations like mutation and crossover: for such operations, the possibility of representing values not in $\mathcal{A}_{n,k}$ is a drawback, as it can require to perform checks in order to determine if values generated are valid codewords.

Moreover, minimizing the memory occupation of a coded k-tree is important for applications storing many such data in a limited amount of memory.

In the following we discuss how codewords can be efficiently represented in roughly $\log(|\mathcal{A}_{n,k}|)$ bits.

First we detail how to represent S, the sequence of pairs. Each pair $(p, \ell) \in [1, n-k] \times [1, k]$ can be easily represented in $\lceil \log(n-k) \rceil + \lceil \log k \rceil$ bits. In order to optimize the space requirement of a single pair, we can represent it as the single integer $(p-1) \cdot k + (\ell - 1)$, thus using $\lceil \log((n-k)k) \rceil$ bits. When $((n-k)k)$ is not a power of two, we can represent the special pair $(0, \varepsilon)$ with any bit sequence not representing a pair in $[1, n-k] \times [1, k]$. Otherwise one more bit is needed. Hence $(n-k-2)\lceil \log((n-k)k+1) \rceil$ bits are required to represent the whole sequence S. Applying the same reasoning we exploited on pairs we can represent S as a single integer, thus the total number of bits can be further reduced to $\lceil (n-k-2) \log((n-k)k+1) \rceil$.

We now discuss several ways to represent $Q \in \binom{[1,n]}{k}$.

The easiest way to represent Q consists in a list of k values in $[1, n]$. This requires $k\lceil \log n \rceil$ bits. Even though n^k has the same asymptotical order of $\binom{n}{k}$, the possibility to represent lists with repetitions is a drawback.

If $k = \Theta(n)$ we can consider to represent Q with its characteristic vector. This requires exactly n bits but still allow us to represent values not in $\binom{[1,n]}{k}$.

A non redundant representation of Q is given by its index in the lexicographically ordered list L of all $X \in \binom{[1,n]}{k}$. In order to efficiently compute this index $id(Q)$ notice that the first $\binom{n-1}{k-1}$ elements in L contain 1, while the remaining $\binom{n-1}{k}$ elements do not contain it. Exploiting this observation we can compute $id(Q)$ with the following recursive function as $id(Q) = \rho(Q, 1, k, n)$, where:

$$\rho(Q, i, k, n) = \begin{cases} 0 & \text{if } k = 0, \\ \rho(Q \setminus i, i+1, k-1, n-1) & \text{if } i \in Q, \\ \binom{n-1}{k-1} + \rho(Q, i+1, k, n-1) & \text{otherwise.} \end{cases}$$

This computation requires $O(nk)$ time since all binomial coefficients can be precomputed with dynamic programming (or with more sophisticate approaches [29]) and each sum between $\binom{n-1}{k-1}$ and $\rho(Q, i+1, k, n-1)$ can be

done in $O(k)$ (these numbers are bigger that $\log n$ bits, then we cannot assume basic operations on them to require constant time).

Exploiting the consideration made on the representation of S and Q we derive that a each k-tree of n nodes can be univocally represented in $\log(|\mathcal{T}_{n,k}|)$ bits.

8 Conclusions

In this paper we introduced a new bijective code for labeled k-trees, moreover we provided coding and decoding algorithms whose running time is linear with respect to the input size.

In order to develop our bijective code for k-trees we exploited 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, among all codes known for Cayley's trees, is motivated by the necessity to identify and discard some redundant information. This is crucial to ensure the resulting code for k-trees to be bijective.

It is worth to notice that our code can be exploited, with minor modifications, to bijectively code Rényi k-trees and arbitrarily rooted k-trees as well.

For Rényi k-trees, it is enough to omit Step 1 of the coding process, and return the string S produced by the Generalized Dandelion Code without removing any redundant pair. The resulting codewords belong to the set $\mathcal{B}_k^n$.

In the case of arbitrarily rooted k-trees, we can assign $Q = R$ in Step 1, without computing l_M. This will have no drawback as we do not need to remove any redundant pair from S in Step 3. The resulting codewords belong to the set $\binom{[1,n]}{k} \times (\{(0,\varepsilon)\} \cup ([1,n-k] \times [1,k]))^{n-k-1}$.

We think that our paper completely closes the problem of efficiently coding and decoding k-trees since we presented linear time algorithms and compact codeword representations. As a future direction for research in this topic, we propose to work on bijective codes for partial k-trees.

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, I. Finocchi, and R. Petreschi. On Coding Labeled Trees. *To appear on Theoretical Computer Science*, 2007.

6. S. Caminiti, E.G. Fusco, and R. Petreschi. A Bijective Code for k-Trees with Linear Time Encoding and Decoding. In *Proceedings of the IntErnational Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07), LNCS 4614*, pages 408–420, 2007.
7. 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.
8. A. Cayley. A Theorem on Trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.
9. W.Y.C. Chen. A general bijective algorithm for trees. *Proceedings of the National Academy of Science, USA*, 87:9635–9639, 1990.
10. W.Y.C. Chen. A Coding Algorithm for Rényi Trees. *Journal of Combinatorial Theory*, 63A:11–25, 1993.
11. Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology.* World Scientific, 2002.
12. N. Deo, N. Kumar, and V. Kumar. Parallel Generation of Random Trees and Connected Graphs. *Congressus Numerantium*, 130:7–18, 1998.
13. 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.
14. W. Edelson and M.L. Gargano. Feasible Encodings For GA Solutions of Constrained Minimal Spanning Tree Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00)*, page 754, Las Vegas, Nevada, USA, 2000.
15. 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.
16. Ö. Eğecioğlu and L.P. Shen. A Bijective Proof for the Number of Labeled q-Trees. *Ars Combinatoria*, 25B:3–30, 1988.
17. D. Foata. Enumerating k-Trees. *Discrete Mathematics*, 1(2):181–186, 1971.
18. C. Greene and G.A. Iba. Cayley's Formula for Multidimensional Trees. *Discrete Mathematics*, 13:1–11, 1975.
19. F. Harary and E.M. Palmer. On Acyclic Simplicial Complexes. *Mathematika*, 15:115–122, 1968.
20. A. Kelmans, I. Pak, and A. Postnikov. Tree and Forest Volumes of Graphs. Technical report, DIMACS 2000-03, 2000.
21. 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.
22. J.W. Moon. The Number of Labeled k-Trees. *Journal of Combinatorial Theory*, 6:196–199, 1969.
23. J.W. Moon. *Counting Labeled Trees.* William Clowes and Sons, London, 1970.
24. E.H. Neville. The Codifying of Tree-Structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.
25. S. Picciotto. *How to Encode a Tree.* PhD thesis, University of California, San Diego, 1999.
26. H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:142–144, 1918.
27. 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.

28. D.J. Rose. On Simple Characterizations of k-Trees. *Discrete Mathematics*, 7:317–322, 1974.
29. I. Vardi. *Computational Recreations in Mathematica*, chapter Computing Binomial Coefficients. Redwood City, CA, 1991.