SAPIENZA
UNIVERSITY OF ROME

On Coding Labeled Trees

Advisor
Prof. Rossella Petreschi

Ph.D. Candidate
Saverio Caminiti

Academic Year 2006/2007

Author Address:

Saverio Caminiti
Computer Science Department
Sapienza University of Rome
Via Salaria 113, I-00198 Rome, Italy
Email: `caminiti@di.uniroma1.it`
Homepage: `caminiti@di.uniroma1.it`

to Beatrice

Abstract

Trees are probably the most studied class of graphs in Computer Science. In this thesis we study bijective codes that represent labeled trees by means of string of node labels. We contribute to the understanding of their algorithmic tractability, their properties, and their applications.

The thesis is divided into two parts. In the first part we focus on two types of tree codes, namely Prüfer-like codes and Transformation codes. We study optimal encoding and decoding algorithms, both in a sequential and in a parallel setting. We propose a unified approach that works for all Prüfer-like codes and a more generic scheme based on the transformation of a tree into a functional digraph suitable for all bijective codes. Our results in this area close a variety of open problems.

We also consider possible applications of tree encodings, discussing how to exploit these codes in Genetic Algorithms and in the generation of random trees. Moreover, we introduce a modified version of a known code that, in Genetic Algorithms, outperform all the other known codes.

In the second part of the thesis we focus on two possible generalizations of our work. We first take into account the classes of k-trees and k-arch graphs (both superclasses of trees): we study bijective codes for this classes of graphs and their algorithmic feasibility. Then, we shift our attention to Informative Labeling Schemes. In this context labels are no longer considered as simple unique node identifiers, they rather convey information useful to achieve efficient computations on the tree. We exploit this idea to design a concurrent data structure for the lowest common ancestor problem on dynamic trees. We also present an experimental comparison between our labeling scheme and the one proposed by Peleg for static trees.

Acknowledgments

I would like to express my gratitude to my advisor Rossella for all she did for me in these years. She suggested me to enroll for a PhD position and she helped me moving my first steps in algorithmic research. She always supported, tolerated, and motivated me, believing in my ability even more than I did sometimes. I am also very grateful to Irene and Tiziana for the time they spent working with me, their advices, and their help, being always kind and friendly.

Special thanks go to Prof. Narsingh Deo and Prof. András Frank. The former gave me the chance to visit the Computer Science Department at the University of Central Florida; the latter made it possible for me to join EGRES group at the ELTE University in Budapest as a visiting researcher. Both these experiences have been fundamental parts of my professional growth.

Explicitly, I want to thank Prof. Alessandro Mei for being in my thesis committee, and the external reviewers Prof. Luisa Gargano and Prof. Ömer Eğecioğlu for their precious comments to this thesis. I would also thank all my further coauthors for the interesting discussions and the work done during these years: Prof. Stephan Olariu, Paulius Micikevičius, Guillaume

Fertin, and Emanuele G. Fusco.

I would also like to say thanks to all my friends that shared with me all the joys and pains of being a student in Rome; Rosa who made the stay in Florida better for me and my wife; and all the guys I knew in Budapest.

Finally, I give thanks to my family for the support they provided me through my entire life and in particular, I must acknowledge my wife – without her love, encouragement and help, I would not have finished this thesis.

Contents

Chapter 1

Introduction

Trees are probably the most studied class of graphs in Computer Science. They are used in a large variety of domains, including computer networks, computational biology, databases, pattern recognition, web mining. In almost all applications, tree nodes and edges are associated with labels, weights, or costs. Examples range from XML data to tree-based dictionaries (heaps, AVL, RB-trees), from phylogenetic trees to spanning trees of communication networks, from indexes to tries (used in compression algorithms). Many data structures can be used to represents trees: adjacency matrices, adjacency lists, parent vectors, and balanced parentheses are just a few examples.

An interesting alternative to the usual representations of tree data structures in computer memories is based on coding labeled trees by means of strings of node labels. String-based codes for labeled trees have many practical applications. For example, they are used in fault dictionary storage [12], distributed spanning tree maintenance [48], generation of random trees [36], Genetic Algorithms [75, 92].

There are codes that define bijections between the set of labeled trees and a set of strings of node labels. In these one-to-one mappings, the length of the string must be equal to $n-2$, since Cayley has proved that the number of labeled trees on n nodes is n^{n-2} [26][1], In his proof of Cayley's theorem, Prüfer

[1] For the sake of correctness we report that Borchardt [13] proved this result almost 30 years before Cayley, in 1860. Cayley independently rediscovered it in 1889. Nowadays this result is universally known as Cayley's Theorem.

provided the first bijective string-based coding for trees [90]. Over the years, many other bijective codes have been introduced [29, 39, 43, 78, 79, 89].

Motivated by the importance of labeled trees, in this thesis we study algorithmic aspects related to bijective tree encodings. We contribute to the understanding of bijective codes for labeled trees, their algorithmic tractability, their properties, and their applications. The thesis is divided into two parts. In the first part we focus on two types of tree codes, named Prüfer-like codes and Transformation codes. We study optimal algorithms for encoding and decoding, both in a sequential and in a parallel setting. Our results in this area close a variety of open problems. We also consider possible applications of tree encodings, discussing how to exploit these codes in Genetic Algorithms and in the generation of random trees.

In the second part of the thesis we focus on two possible generalizations of our work. We first take into account the class of k-trees [57] (a superclass of trees): we study bijective codes for this class of graphs and their algorithmic feasibility for rooted, unrooted, and Rényi k-trees [94]. Then, we shift our attention to Informative Labeling Schemes [87]. In this context labels are no longer considered as simple unique node identifiers, they rather convey information useful to achieve efficient computations on the tree. We exploit this idea to design a concurrent data structure for the lowest common ancestor problem on dynamic trees. In the literature other generalizations and specializations of tree codes not considered in this thesis have been studied [44, 45, 83, 93].

Let us now detail the content of each chapter. We defer to Section 1.1 for a description of all original contributions of this thesis. We refer the reader to the Glossary at the end of this thesis for all notations and definitions not explicitly introduced elsewhere.

Part I: Tree Encodings

In Chapter 2 we recall the Prüfer code as introduced in [90] and Prüfer-like codes that hinge upon the same fundamental idea, i.e., recursive elimination of leaves. Prüfer-like codes are due to Neville [79], Deo and Micikevičius [39]. We discuss how these codes can be used both for rooted and unrooted trees.

In Chapter 3 we initially survey known optimal algorithms for encoding and decoding Prüfer-like codes. Then, we introduce a unified approach that works for all of them [17, 20]. By means of our unified approach we completely close the problem of encoding and decoding all these codes in a sequential setting. We also provide efficient parallel algorithms that either match or improve the performances of the best previous known results.

In Chapter 4 we describe two possible applications of tree encodings: random trees generation and Genetic Algorithms. The first application shows how these combinatorial bijections can be fruitfully exploited to guarantee that trees are generated uniformly at random, both in sequential and parallel settings. We also present an experimental analysis showing that this method is competitive with other known methods. Genetic Algorithms provide a wider example of application of tree encodings. Many experimental comparisons have been presented in the literature, exploring several possible tree encodings. Some of these experiments shifted our attention from Prüfer-like code to Transformation codes.

In Chapter 5 we focus on bijective codes not belonging to the class of Prüfer-like codes. We approaches proposed by Eğecioğlu and Remmel [43] and by Picciotto [89], providing a general scheme based on the transformation of a tree into a functional digraph (from which the name Transformation codes). By means of our general scheme we are able to compare the codes and provide theoretical reasons for their performances in Genetic Algorithm implementations [24, 25].

Part II: Generalizations

In Chapter 6 we consider the class of k-trees, a natural generalization of trees [57], and study bijective codes for labeled k-tree. We survey known results and introduce a novel code together with encoding and decoding algorithms. The running time of our algorithms is linear with respect to the size of the encoded k-tree. Our code can be easily adapted to rooted, unrooted, and Rényi k-trees, preserving bijectivity [22, 23]. We conclude the chapter with some considerations on the number of k-arch graphs (a superclass of k-trees): we consider enumerative results for this class given by Lamathe [72], we prove that one such result is erroneous and provide a suitable correction [21].

In Chapter 8 we present the concept of Informative Labeling Scheme (ILS) introduced by Peleg [87] and propose an ILS for Lowest Common Ancestor on dynamic trees. We exploit it to obtain a concurrent data structure for LCA of dynamic trees [18]. We also experimentally compare our scheme with the one proposed by Peleg and show pros and cons of both schemes [19].

1.1 Original Contributions of this Thesis

A Unified Approach for Prüfer-like Codes. The unified approach presented in Chapter 3 makes it possible to encode and decode all Prüfer-like codes introduced by Prüfer [90], Neville [79], Deo and Micikevičius [39]. The unified encoding algorithm is based on the definition of pairs associated to tree nodes according to criteria dependent on the specific code: the coding problem is then reduced to the problem of sorting these pairs in lexicographic order. The unified decoding algorithm hinges upon the computation of the rightmost occurrence of each value in a codeword. By exploiting this approach, we obtain optimal linear time algorithms for encoding and decoding all Prüfer-like codes presented in Chapter 2. We close the open problem of finding a linear time decoding algorithm for the Second Neville code. We also show how it is possible to parallelize our unified approach: our unified algorithms either match or improve by a factor $\sqrt{\log n}$ the performances of the best *ad hoc* parallel algorithms known so far.

These results have been published on *Theoretical Computer Science* [20], in the special issue devoted to the 6^{th} *Latin American Symposium on Theoretical Informatics (LATIN'04)* where a preliminary version of this work appeared [17]. A part of these results also appeared in *Congressus Numerantium* [16].

A General Scheme for Transformation Codes. In Chapter 5 we introduce a general scheme for defining bijective codes based on the transformation of a tree into a functional digraph. The class of Transformation codes (i.e., those codes that can be defined with our general scheme) contains each possible bijective code for labeled trees. The same is not true for other classes, such as Prüfer-like codes. As examples, we show how it is possible to map the codes by Eğecioğlu and Remmel [43] and by Picciotto [89] into our scheme. This also gives us a better comprehension of how encoding preserves the topology of the tree, and therefore helps to understand which code better fits some desirable properties, such as locality and heritability.

These results have been published in *Proceedings of the* 11^{th} *International Conference on Computing and Combinatorics (COCOON'05)* [25]. An extended version of this work has been submitted to *SIAM Journal of Discrete Mathematics* [24]. A part of these results also appeared in *Congressus Numerantium* [16].

Optimal Algorithms for k-Trees Encoding. In Chapter 6 we introduce a novel bijective code for rooted and unrooted k-trees (and also for Rényi k-trees). We give a detailed description of linear time encoding and decoding algorithms for our code. We also analyze, in Chapter 7, the result presented by Lamathe [72] concerning the number of k-arch graphs. We point out an error in his work: the closed formula he provided overestimates the cardinality of this class of graphs. We provide an exact counting result in terms of a recursive function.

The results concerning k-trees encoding have been published in *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)* [23]. An extended version of this work has been submitted to *Theory of Computing Systems* [22]. The correct formula for the number of k-arch graphs appeared on *Journal of Integer Sequences* [21].

Informative Labeling Schemes for LCA on Dynamic Trees. In Chapter 8 we exploit the idea of Informative Labeling Schemes to design concurrent data structures. The scenario we have considered is a multiprocessor machine with asynchronous access to a shared memory. We focus on a data structure for the Lowest Common Ancestor Problem on dynamic trees. We propose a new Informative Labeling Scheme for Lowest Common Ancestor that may be used for dynamic trees and show a detailed experimental comparison between our scheme and the one proposed by Peleg [87] for LCA on static trees. These results have not yet been published [18, 19].

Part I

Tree Encodings

Chapter 2

Prüfer-Like Codes

We start this chapter recalling the well known Prüfer code [90], originally introduced by the German mathematician in 1918 to provide an alternative proof of Cayley's theorem.

The Prüfer code deals with unrooted trees on n nodes labeled with distinct values from a set L of cardinality n. Such trees are known as *Cayley trees.* Without loss of generality we will assume that the labels are integer numbers in the range from 1 to n, i.e., that $L = [1, n]$. Moreover, we will identify a node with its label: the node set of a tree on n nodes will therefore be $[1, n]$. The set of Cayley trees on n nodes is denoted as $\mathcal{T}_n$. It is well known that $|\mathcal{T}_n| = n^{n-2}$:

Theorem 2.1 (Cayley [26]). *There exist n^{n-2} unrooted trees with n nodes univocally labeled with n distinct labels.*

Let us call $\mathcal{R}_n$ the class of rooted Cayley trees with n nodes. Since there are n different possible ways to root a Cayley tree in $\mathcal{T}_n$, it directly follows that $|\mathcal{R}_n| = n|\mathcal{T}_n|$. More formally, from Theorem 2.1 we obtain:

Corollary 2.2. *There exist n^{n-1} rooted trees with n nodes univocally labeled with n distinct labels.*

We continue the chapter surveying other codes that hinge upon the same fundamental idea exploited by Prüfer: for this reason they are called Prüfer-like codes. The codes we will study are due to Neville [79] and to Deo and Micikevičius [39].

This chapter does not contain any original contribution but it is helpful to understand the followings. It is organized as follows: in Section 2.1 we recall the original encoding and decoding processes as introduced by Prüfer [90]; we also clarify how to apply it to rooted trees. In Section 2.2 and Section 2.3 we describe the Prüfer-like codes introduced by Neville [79] and by Deo and Micikevičius [39], both for rooted and unrooted trees.

We will use $adj(v)$ to refer to the set of all the nodes adjacent to any node v. If $adj(v)$ consists of a single node (i.e., v is a leaf), when there is no ambiguity, we will use $adj(v)$ to refer to the adjacent node, itself rather then to a set of cardinality 1. Let T be a tree and v a leaf in T, we denote $T \setminus v$ the tree obtained form T by removing the node v and the edge incident on node v.

2.1 Prüfer Code

The Prüfer code is a bijective association between trees in $\mathcal{T}_n$ and sequences of $n-2$ node labels. We will say that the sequence associated to a tree T is the *codeword* for T. Codewords are strings of length $n-2$ over the alphabet $[1,n]$ and thus belong to $[1,n]^{n-2}$. The operator :: will be used to denote the concatenation of two strings.

The original definition of the Prüfer code was formulated in terms of a recursive elimination of leaves. Given an unrooted tree T, recursively remove the smallest leaf until a single node remains. Let a_i be i-th removed leaf and b_i the node adjacent to a_i when a_i has been removed. The sequence:

$$\begin{pmatrix} a_1, a_2, \ldots, a_{n-1} \\ b_1, b_2, \ldots, b_{n-1} \end{pmatrix} \tag{2.1}$$

univocally describes T, since each edge of the tree appears as a pair (a_i, b_i) for some i. This sequence of pairs is known as the *Natural Code* for T. Notice that b_{n-1} is always n, since this node will never be selected as a smallest leaf to be removed.

The Prüfer code for T is the string:

$$C = (b_1, b_2, \ldots, b_{n-2})$$

The following interesting property holds:

Property 2.3. *Given an unrooted Cayley tree T, let C be its Prüfer code. Each node v of T appears in C exactly $deg(v) - 1$ times.*

Proof. Consider the Natural Code for T. Since Equation 2.1 is a list of all edges in T, each node v appears in the Natural Code exactly $deg(v)$ times. Any node but n appears in the sequence $a_1, \ldots, a_{n-1}$ exactly once, while n does not appear in this sequence: n appears in $b_1, \ldots, b_n$ exactly $deg(n)$ times and $b_n = n$. This implies that each node appears in $C = (b_1, \ldots, b_{n-1})$ exactly $deg(v) - 1$ times. □

Before showing how to invert this bijection (i.e., how to rebuild a tree from its codeword), let us introduce a more formal definition for the code. We will consider the encoding procedure of the Prüfer code as a recursive function π:

$$\pi(T) = adj(min_T) :: \pi(T \setminus min_T) \tag{2.2}$$

where min_T denotes the minimum leaf in T. If T has only 2 nodes $\pi(T)$ is the empty string.

Example 1. In Figure 2.1 an example of coding is provided. At the beginning the smallest leaf is $a_1 = 3$, $adj(3) = \{2\}$ and then $b_1 = 2$. Once node 3 is removed from the tree, node 2 the smallest leaf: $a_2 = 2$ and its adjacent node is $b_2 = 5$, then node 2 is removed. At the next steps $a_3 = 4$, $b_3 = 1$, $a_4 = 6$, $b_4 = 1$. As soon as node 6 is removed node 1 becomes a leaf and then $a_5 = 1$ and $b_5 = 5$. The process stops when the tree consists of a single edge $(5, 7)$ since a codeword of length $n - 2$ has been computed. The Prüfer code for this tree is $(2, 5, 1, 1, 5)$.

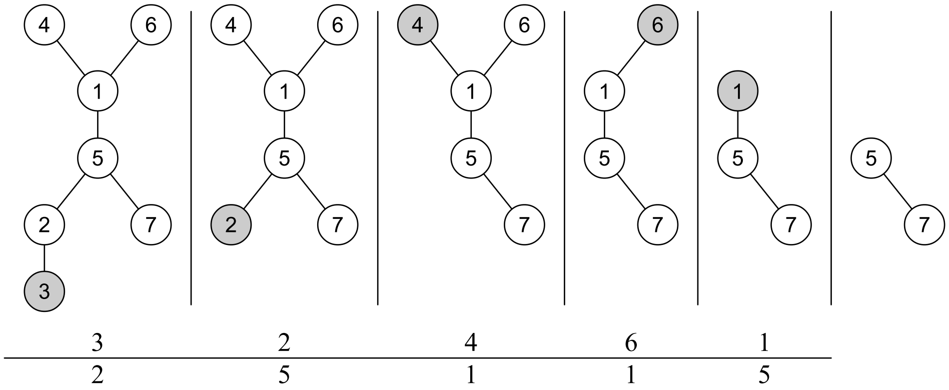


Figure 2.1: Step by step computation of Prüfer code. At each step the leaf with smallest label is deleted. The resulting codeword is $(2, 5, 1, 1, 5)$.

2.1.1 Decoding

We now show how it is possible to decode Prüfer code. Given any codeword C in $[1, n]^{n-2}$, the decoding computes a tree T such that $\pi(T) = C$.

From Property 2.3 we know that leaves in T are all nodes that do not appear in $C = (b_1, \ldots, b_{n-2})$. We can therefore compute a_1 as the smallest number in $[1, n]$ not in C. In order to compute a_2 consider that $(b_2, \ldots, b_{n-2})$ is the codeword for $T' = T \smallsetminus \{a_1\}$: indeed function π recursively computes $(b_2, \ldots, b_{n-2})$ as $\pi(T')$ (see Equation 2.2). Thus we can identify the smallest leaf in T' as the smallest number in $[1, n] \smallsetminus \{a_1\}$ not in $(b_2, \ldots, b_{n-2})$. Analogously $(b_3, \ldots, b_{n-2}) = \pi(T' \smallsetminus \{a_2\})$, and then a_3 is the smallest number in $[1, n] \smallsetminus \{a_1, a_2\}$ not in $(b_2, \ldots, b_{n-2})$. In general a_i is the smallest number in $[1, n] \smallsetminus \{a_1, \ldots, a_{i-1}\}$ not in $(b_i, \ldots, b_{n-2})$.

In order to complete the reconstruction of the Natural Code consider that b_{n-1} is always n and a_{n-1} should be the only number smaller than n not yet used in $(a_1, \ldots, a_{n-2})$. The tree is $T = ([1, n], \{(a_i, b_i) : i \in [1, n-1]\})$.

It is easy to see that the decoding procedure is the inverse function of π, moreover π is injective and surjective: the Prüfer code is a bijection between $\mathcal{T}_n$ and $[1, n]^{n-2}$. For formal proofs of these assertions we refer the interested reader to [90]. In these thesis we rather focus on algorithmic implications.

Before showing how to extend the Prüfer code to rooted trees, let us show an example of decoding.

Example 2. Given the codeword $(2,5,1,1,5)$ we can easily deduce $n = 7$ since the codeword has length 5. The set of all leaves of the encoded tree T is $\{3,4,6,7\}$, i.e., all those numbers in $[1,7]$ not appearing the codeword. The smallest leaf is $a_1 = 3$. Leaves of $T \setminus \{3\}$ are all those nodes in $[1,7] \setminus \{3\}$ not appearing in the sequence $(5,1,1,5)$, i.e., $\{2,4,6,7\}$. Then the leaf removed at the second step of the encoding should be $a_2 = 2$. The algorithm proceeds by choosing $a_3 = \min([1,7] \setminus \{2,3\} \setminus (1,1,5)) = 4$, $a_4 = \min([1,7] \setminus \{2,3,4\} \setminus (1,5)) = 6$, and $a_5 = \min([1,7] \setminus \{2,3,4,6\} \setminus (5)) = 1$. To complete the reconstruction of the Natural Code for T the algorithm chooses $b_6 = 7$ and $a_6 = 5$, i.e., the only number in $[1,6]$ not yet chosen as a_i. The Natural Code obtained univocally identifies the tree depicted in Figure 2.1:

$$\begin{pmatrix} 3,2,4,6,1,5 \\ 2,5,1,1,5,7 \end{pmatrix}$$

2.1.2 Rooted Trees

The encoding procedure proposed by Prüfer can be applied to a rooted tree. In this case the root is never considered as a leaf, even if it has degree 1, and is never removed from the tree during the encoding. This implies that, in the Natural Code, b_{n-1} is the tree root. Since this information changes according to the encoded tree, it cannot be omitted. Thus, the codeword has length $n-1$ and the code a bijection between $\mathcal{R}_n$ and $[1,n]^{n-1}$.

We remark that Property 2.3 slightly changes when the code is applied to rooted trees:

Property 2.4. *Given a Cayley tree T rooted at r, let C be its Prüfer code. The root r appears in C exactly deg(r) times and each other node v of T appears in C exactly deg$(v) - 1$ times.*

Example 3. In Figure 2.2 an example of encoding a rooted tree is presented. At each step the smallest leaf is removed and its parent (i.e., its unique adjacent node) is added to the codeword. We remember that the root is

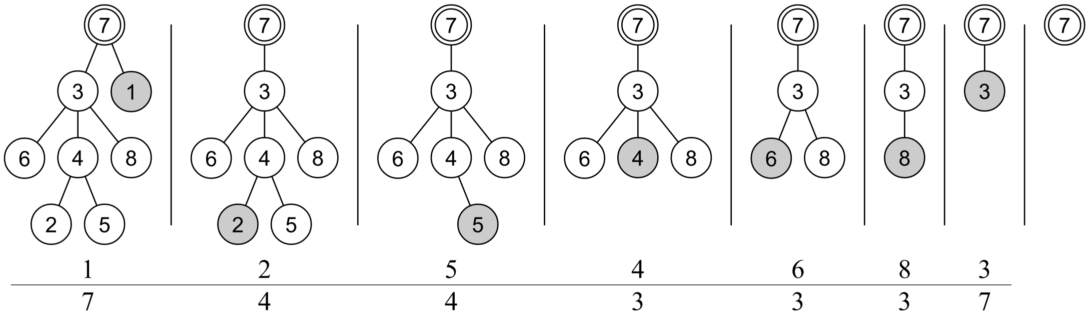


Figure 2.2: Step by step computation of Prüfer code for a rooted tree. At each step the leaf with smallest label is deleted. The resulting codeword is $(7, 4, 4, 3, 3, 3, 7)$.

never considered as a leaf, even if it has degree 1. The resulting $n-1$ length codeword is $(7, 4, 4, 3, 3, 3, 7)$.

The decoding procedure simply deduces the sequence of the $n-1$ removed leaves, as described for unrooted trees. Then, it reconstructs the Natural Code, and returns the corresponding tree rooted in the last symbol of the codeword.

We can also deal with Cayley trees rooted in a fixed node x (such as the node 1, the node with maximum label, the node with label $\lceil\sqrt{n}\,\rceil$, etc.), let us call $\mathcal{T}_n^x$ the class of such trees. In this case the codeword length can be reduced to $n-2$: there is no need to maintain the last element b_{n-1} since it is always x. The code is bijective because, for each x, $|\mathcal{T}_n^x| = n^{n-2}$.

It is worth noticing that the codeword that the Prüfer encoding procedure associates to a tree $T \in \mathcal{T}_n^n$ is exactly the same codeword obtained by the original Prüfer code applied to T as an unrooted tree.

2.2 Neville's Codes

In 1953, Neville [79] presented three different codes. The first one coincides with Prüfer code, while the other two constitute novel bijections between Cayley trees and strings of node labels. Remarkably, all of them have been described in terms of recursive leaves elimination; each time a leaf is removed from the tree, the unique node adjacent to the leaf is added to the codeword. Because of this similarity with the Prüfer code these codes are called *Prüfer-like* codes. However, each code has a specific criterion to determine the sequence of leaves eliminated at each step.

In the following we will describe all codes as applied to rooted Cayley trees. To apply these codes to unrooted trees, as stated in Section 2.1.2, it is enough to root the tree in a fixed node (e.g., always n) and omit the last symbol in the codeword. Indeed Neville's codes have been originally introduced for trees rooted in the fixed node n, but have been later generalized by Moon [78] to unrooted and arbitrarily rooted trees. All these codes, similarly to the Prüfer code, satisfy Properties 2.3 and 2.4.

2.2.1 Second Neville Code

The Second Neville code, at each step, removes from the tree all the leaves in increasing label order. The parent (the unique adjacent node) of each leaf is added to the codeword.

Example 4. In Figure 2.3 an example of tree encoding with the Second Neville code is shown. At the first step all leaves $\{8, 3, 5, 9, 4\}$ are removed in increasing label order: $(3, 4, 5, 8, 9)$. The first five symbols in the codeword corresponds to $(6, 10, 6, 1, 7)$, i.e., the labels of nodes adjacent to removed leaves. Iterating this process until the tree consists of a single node a codeword of length $n - 1$ is obtained: $(6, 10, 6, 1, 7, 2, 7, 7, 7)$.

The process of decoding the Second Neville code is analogous to the one described in Section 2.1.1, except for the fact that, at each step, rather than the smallest leaves we chose all leaves in increasing order. Let us show an example of decoding the codeword $C = (6, 10, 6, 1, 7, 2, 7, 7, 7)$ to obtain a rooted tree.

Example 5. The codeword length is $n - 1$ then $n = 10$. The set of leaves of the initial tree is $\{3, 4, 5, 8, 9\}$, i.e., all values not in C. These values, together with the first 5 symbols in C, allow us to deduce the edges $(3, 6)$, $(4, 10)$, $(5, 6)$, $(8, 1)$, and $(9, 7)$. The remaining part of the codeword $(2, 7, 7, 7)$ corresponds

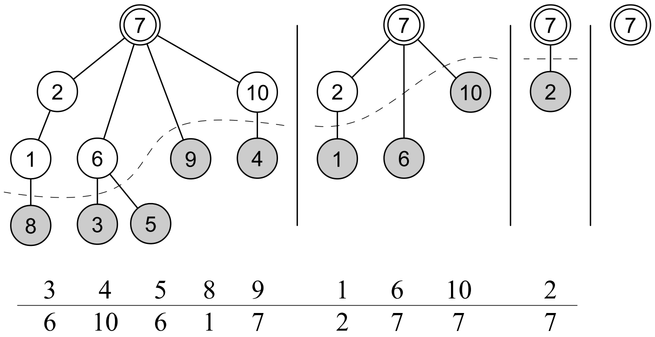


Figure 2.3: Step by step computation of the Second Neville code. At each step all leaves are deleted in increasing label order. The resulting codeword is $(6, 10, 6, 1, 7, 2, 7, 7, 7)$.

to the encoding of the subtree whose nodes are $[1, 10] \smallsetminus \{3, 4, 5, 8, 9\}$. All these nodes, but 2 and 7 that appear in the codeword, are leaves: $\{1, 6, 10\}$. The edges identified are: $(1, 2)$, $(6, 7)$, and $(10, 7)$. In the last step the codeword consists of a single symbol (7) and the subtree represented by this codeword has only two nodes: $\{2, 7\}$. Then the last edge is $(2, 7)$.

We recall that, as noted in Section 2.1.2, the last symbol of the codeword is the root of the tree. Moreover, each edge deduced during the decoding process can be created as an oriented edge going from a node (the leaf) to its parent (the node in the codeword).

We want to remark that when the Second Neville code is applied to unrooted trees (as done by Moon in [78]) the last node remaining in the tree after the encoding is the center of the tree. It corresponds to the last symbol in the codeword (i.e., the one that should be omitted to obtain an $n - 2$ length codeword). If the tree has two centers the one with highest label will be the last node.

2.2.2 Third Neville Code

The Third Neville code at the first step selects the smallest leaf and removes it. In the subsequent steps, if the node adjacent to the last removed leaf is now a leaf, it is selected and removed; otherwise the new smallest leaf is

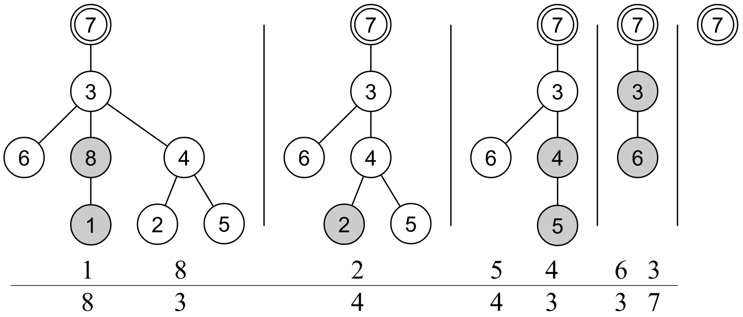


Figure 2.4: Step by step computation of the Third Neville code. At each step the pending chain containing the leaf with smallest label is deleted. The resulting codeword is $(8, 3, 4, 4, 3, 3, 7)$.

selected. In other words, nodes that are not leaves in the initial tree are removed as soon as they become leaves, while all leaves of the initial tree are selected in increasing order.

The example in Figure 2.4 helps us to clarify the criterion.

Example 6. Initially we select leaf 1: when it is removed, its adjacent node 8 becomes a leaf and then it is suddenly chosen and removed. Node 3 still has other children, and thus we seek for a new smallest leaf: 2. The removal of node 2 does not let its parent 4 becomes a leaf, and then the smallest leaf 5 is chosen. Once 5 is removed 4 becomes a leaf and is removed. The only remaining leaf is 6 whose removal lets 3 become a leaf. Finally 3 is removed. As in Prüfer code, each time a leaf is removed, the label of its adjacent node is added to the codeword, then the Third Neville code for this tree is $(8, 3, 4, 4, 3, 3, 7)$.

An alternative way to look at this code is as it works by deleting chains. We call *pending chain* a path $u_1, \ldots, u_k$ of maximal length such that the starting point u_1 is a leaf, and, for each $i \in [1, k-1]$, the deletion of u_i makes u_{i+1} a leaf: the code works by iteratively deleting the pending chain containing the smallest leaf.

Moon in [78] applied the Third Neville code to unrooted trees. In this case the last node remaining in the tree after the encoding is the leaf with maximum label. Indeed, consider the set of all leaves of the initial tree in

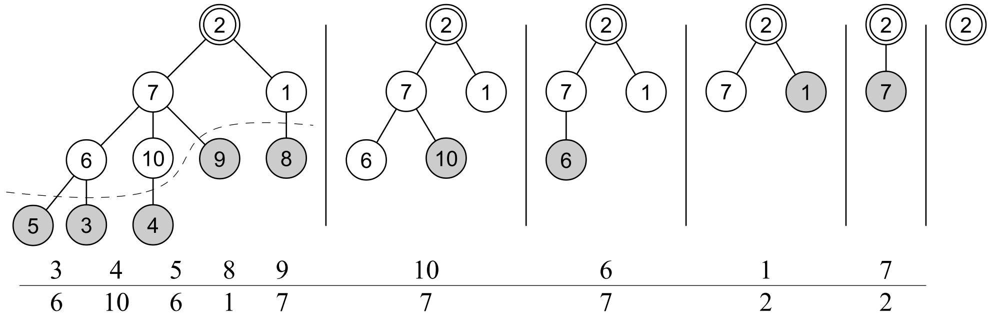


Figure 2.5: Step by step computation of the Stack-Queue code. At the first step all the leaves are deleted in increasing label order, other nodes are deleted in the order in which they become leaves. The resulting codeword is $(6, 10, 6, 1, 7, 7, 7, 2, 2)$.

increasing order $\{l_1, l_2, \ldots, l_k\}$. After the removal of pending chains corresponding to $l_1, l_2, \ldots, l_{k-2}$, the tree consists of a single chain joining l_{k-1} to l_k. This chain is removed starting from l_{k-1}, then the last node must be l_k.

2.3 Stack-Queue Code

Recently, Deo and Micikevičius [39] introduced a new Prüfer-like code called Stack-Queue code. This code initially deletes all tree leaves in increasing label order, as the Second Neville code. Then, it deletes all the internal nodes in the order in which they become leaves. The original presentation makes use of specific data structures. At the beginning of the encoding, a FIFO queue Q is initialized with all tree leaves in increasing label order. At each step the algorithm extracts a leaf from Q and removes it. Whenever a node becomes a leaf, it is added to Q. We defer to Chapter 3 the explicit presentation of their algorithm.

Example 7. Consider the tree in Figure 2.5. Let us explicitly report the content of the queue Q at each step of the algorithm. It initially contains all leaves in increasing order: $Q^{(0)} = (3, 4, 5, 8, 9)$; in the following steps the queue changes as follows: $Q^{(1)} = (4, 5, 8, 9)$, $Q^{(2)} = (5, 8, 9, 10)$, $Q^{(3)} = (8, 9, 10, 6)$, $Q^{(4)} = (9, 10, 6, 1)$, $Q^{(5)} = (10, 6, 1)$, $Q^{(6)} = (6, 1)$, $Q^{(7)} = (1, 7)$, $Q^{(8)} = (7)$, $Q^{(9)} = ()$. The leaves elimination order is given by the sequence of nodes extracted from the queue: $3, 4, 5, 8, 9, 10, 6, 1, 7$. The resulting codeword is $(6, 10, 6, 1, 7, 7, 7, 2, 2)$.

The original decoding algorithm proceeds backwards and uses a LIFO stack S. As a first step the codeword C is scanned right to left and each value is pushed in S (avoiding duplicates). Then all leaves (values not in C) are pushed in S in decreasing label order. This ensures that S contains all nodes that have been pushed to Q during the encoding in the reverse order. Therefore, popping values out of S, we have the exact leaves elimination order realized by the encoding. This correctly reconstructs the tree.

Example 8. For example, decoding the codeword $C = (6, 10, 6, 1, 7, 7, 7, 2, 2)$ will make the content of the stack to be $S = (2, 7, 1, 6, 10)$ after the right to left scan of C. Then adding all values not in C in decreasing order we obtain $S = (2, 7, 1, 6, 10, 9, 8, 5, 4, 3)$. Popping $n-1$ elements out of S we obtain the same leaves elimination order realized by the encoding: $3, 4, 5, 8, 9, 10, 6, 1, 7$. Then we can rebuild the tree.

This code has been originally introduced for unrooted trees. In this case the last node is the center of the tree (either of them if the tree has two centers). This makes it possible to compute the tree diameter directly from the codeword, as shown in [39]. The same property holds for the Second Neville code.

2.4 Concluding Remarks

Up to now, we have recalled four well known Prüfer-like codes. All of them are based on recursive leaves elimination and realize bijection between rooted and unrooted Cayley trees an codewords of length $n-2$ and $n-1$, respectively.

In concluding this chapter, we want to highlight that the criterion used by Prüfer to select the order in which tree nodes are removed (at each step remove the smallest leaf) can be substituted by many other deterministic criterions. The other codes we have seen in this chapter are just three possible examples.

In general, any deterministic criterion can be used to generate a bijective Prüfer-like code, provided that it select, at each step of the encoding, a (non-empty) sequence of the tree leaves exploiting only the current leaves set and the sequences chosen in the previous steps. Indeed, the decoding scheme proposed in Section 2.1.1, at each step, can identify the set of leaves removed in the corresponding step of the encoding simply using the same deterministic criterion.

Chapter 3

Algorithms for Prüfer-like Codes

In this chapter we focus on algorithmic aspects related to the computation of all Prüfer-like codes presented in Chapter 2. Initially, in Section 3.1, we survey a series of *ad hoc* algorithms for Prüfer code, Neville's codes, and Stack-Queue code. All these algorithms strongly depend on the properties of the code which has to be computed and thus are very different from each other.

As a novel contribution of this thesis we present a unified approach that makes it possible to encode and decode all Prüfer-like codes introduced so far. It has been published in [20] (a preliminary version of this work appeared in [17]). The UNIFIED ENCODING ALGORITHM presented in Section 3.2 is based on the definition of pairs associated to tree nodes according to criteria dependent on the specific code: the coding problem is then reduced to the problem of sorting these pairs in lexicographic order. The UNIFIED DECODING ALGORITHM presented in Section 3.3 hinges upon the computation of the rightmost occurrence of each value in a codeword. With these unified algorithms we obtain optimal linear time algorithms for encoding and decoding all Prüfer-like code seen in Chapter 2. It should be noted that we close the open problem of finding a linear time decoding algorithms for the Second Neville code.

Finally, in Section 3.4 we show how it is possible to parallelize our unified approach achieving very good results: our unified algorithms either match or improve the performances of the best *ad-hoc* parallel algorithms known so far. Namely we obtain parallel encoding algorithms that require $O(n)$ operations for the Prüfer code and the Third Neville code and $O(n\sqrt{\log n})$ for the Second Neville code and the Stack-Queue code. Concerning decoding we match the $O(n \log n)$ bound known for Prüfer code and, for the first time, we provide parallel algorithms for the Second Neville code, Third Neville code, and the Stack-Queue code: these algorithms require $O(n\sqrt{\log n})$ operations.

3.1 Known Algorithms

Here we recall known optimal sequential algorithms to encode and decode Prüfer code, Second Neville code, Third Neville code, and Stack-Queue code. We assume to deal with rooted trees.

3.1.1 Prüfer Code

A straightforward implementation of the idea described in Section 2.1 would require to compute, at each of $n-1$ steps, the minimum among a set. Even using appropriate data structures, like a minimum heap, this would lead to algorithms whose running time is $O(n \log n)$.

Since the introduction of Prüfer code in 1918, a linear time algorithm for encoding a tree has been given for the first time only in the late 70's. In fact, it was left as an exercise both in [80] (exercise 46, page 293), and in [40] (exercise 2, page 666). Maybe this is the reason why it has been rediscovered several times, and still nowadays optimal algorithms for Prüfer code appear to be not known by researchers (see for example [42, 51, 61]).

The linear time encoding algorithm that we propose here is the version published in [17] and in [20] of an idea that, according to our knowledge, is due to Klingsberg [68].

Let us assume that the tree is represented by means of adjacency lists and that the degree of each node is known (otherwise, it can be easily computed with a simple scan of the adjacency lists). The input is an unrooted tree represented by adjacency lists. The Prüfer code of T can be computed as follows:

```
1. for each node $v = 1$ to $n$ do
2.   if $deg(v) = 1$ and $v \neq root$ then
3.     let $u$ be the unique node in $adj(v)$
4.     append $u$ to the code and decrease its degree by $1$
5.     while $deg(u) = 1$ and $u < v$ and $v \neq root$ do
6.       let $z$ be the unique node in $adj(u)$
7.       append $z$ to the code and decrease its degree by $1$
8.       $u \leftarrow z$
```

The idea is to consider all nodes in increasing order (variable v in the algorithm), once a leaf is encountered it is selected for removal. Each removal may let at most one node becomes a leaf (variable u). If u becomes a leaf and has a label smaller than v, it will certainly be the smallest leaf, than it is selected for removal. Removing u may let another node becomes a leaf, thus implying a cascading effect: the inner while loop ensures that this problem is handled correctly.

The algorithm terminates when the codeword reaches the desired length ($n - 1$ for rooted trees). In order to achieve $O(n)$ running time the explicit removal of nodes from the tree is avoided. We simply decrease the degree of a node each time a leaf adjacent to it is selected for removal. This allows us to avoid expensive changes in adjacency lists. In this case line 3 requires a scan of the adjacency list of v to identify the unique node not yet removed (removed nodes can be marked with a flag). Each adjacency list is scanned at most once, then the overall running time is linear.

Concerning decoding, a similar idea can be exploited to reconstruct a tree from a codeword. The linear time decoding algorithm due to Klingsberg has been explicitly published in [40]. We present here a slightly modified version. Initially T is a graph with $n = |C| + 2$ nodes and no edges. A symbol n is added at the end of the codeword to ensure that all $n - 1$ edges are correctly computed. We preliminarily mark all nodes that do not appear in C: these are candidate leaves. The algorithm works as follows:

```
for each $v = 1$ to $n - 1$ do
  if $v$ is marked then
    $u = pop(C)$ and add edge $(v, u)$ to $T$
    if $u$ no longer appears in $C$ then mark $u$
    while $u$ is marked and $u < v$ do
      $z = pop(C)$ and add edge $(u, z)$ to $T$
      if $z$ no longer appears in $C$ then mark $z$
      $u \leftarrow z$
```

The operation $pop(C)$ extracts the first symbol from the codeword. To test if a certain node no longer appears in C in $O(1)$ time we can precompute the last occurrence of each value $i \in [1, n]$ in C with a simple right to left scan of the codeword. This is enough to conclude that the running time of this algorithm is $O(n)$.

3.1.2 Second Neville Code

A trivial implementation of the Second Neville code described in Section 2.2.1 would require to sort a set of integer numbers at each step. Using integer sorting algorithms this requires $O(n^2)$ running time. In [38] an encoding algorithm that uses a set of sorted lists is presented. Due to the use of sorted lists, the running time decreases to $O(n \log n)$. The first linear time algorithm for Second Neville code has been presented in [74] and is analogous to the one obtained with the unified encoding algorithm described in Section 3.2.

There were no optimal algorithms for decoding the Second Neville code known in the literature before the one obtained through our unified decoding algorithm described in Section 3.3.

3.1.3 Third Neville Code

A linear time algorithm for computing the Third Neville code is not difficult to obtain. Each step corresponding to the elimination of an internal node does not imply any global computation on the tree (such as identify a minimum or sort a set of nodes), thus each step requires constant time. To efficiently identify the new smallest leaf, when required, it is enough to precompute a list of all leaves of the initial tree in increasing order, this may be done in linear time with any integer sorting algorithm [35]. Analogous considerations hold for decoding.

The Third Neville code can be computed with an algorithm similar to the one presented for Prüfer code in Section 3.1.1, by just omitting the test $u < v$: this guarantees that internal nodes are removed as soon as they become leaves.

```
1. for each node $v = 1$ to $n$ do
2.   if $deg(v) = 1$ and $v \neq root$ then
3.     let $u$ be the unique node in $adj(v)$
4.     append $u$ to the code and decrease its degree by 1
5.     while $deg(u) = 1$ and $v \neq root$ do
6.       let $z$ be the unique node in $adj(u)$
7.       append $z$ to the code and decrease its degree by 1
8.       $deg(u) = 0$
9.       $u \leftarrow z$
```

Line 8 avoids that a node u already used verifies the test in line 2 and contributes to the codeword again.

The decoding algorithm can be obtained similarly from the Prüfer decoding algorithm by removing the test $u < v$. Also in this case, this implies that labels appearing in the codeword are used as soon as they are marked as candidate leaves. It is as follows:

```
for each $v = 1$ to $n$ do
    if $v$ is marked then
        $u = pop(C)$ and add edge $(v, u)$ in $T$
        if $u$ no longer appears in $C$ then mark $u$
        while $u$ is marked do
            $z = pop(C)$ and add edge $(u, z)$ in $T$
            if $z$ no longer appears in $C$ then mark $z$
            unmark $u$
            $u \leftarrow z$
```

3.1.4 Stack-Queue Code

As mentioned in Section 2.3 Deo and Micikevičius, in their original presentation of the Stack-Queue code, provided linear time algorithms. The encoding algorithm uses a FIFO queue Q, while the decoding algorithm uses a LIFO stack S.

The encoding algorithm is the following:

```
for each node $v = 1$ to $n$ except the root do
    if $deg(v) = 1$ then $enqueue(v, Q)$
while $Q$ is not empty do
    $v \leftarrow dequeue(Q)$
    let $u$ be the parent of $v$
    append $u$ to the code and decrease its degree by 1
    if $deg(u) = 1$ and $u$ is not the root then $enqueue(u, Q)$
```

$enqueue(v, Q)$ adds value v to the tail of queue Q, while $dequeue(Q)$ extracts a value from the head of Q.

The decoding algorithm is the following:

```
for each value $v = 1$ to $n$ do
    $used[i]$ = false
```

```
for $i = n - 1$ to 1 do
    if not $used[C[i]]$ then
        $push(i, S)$
        $used[C[i]]$ = true
$push$ all unused values in $S$ in increasing order
for $i = 1$ to $n - 1$ do
    $v \leftarrow pop(S)$
    add edge $(C[i], v)$ in $T$
```

push(i, S) inserts value i into the to of stack S, while *pop*(S) extracts an value from the top of S.

Both these algorithms require linear time [39].

3.2 A Unified Encoding Algorithm

As we said, sequential and parallel encoding and decoding algorithms have been presented in the literature [27, 38, 39, 53, 54, 104], but all of them strongly depend on the properties of the code which has to be computed and thus are very different from each other.

In this section we show a unified approach that works for all Prüfer-like codes introduced so far. Through this unified approach we obtain linear time coding and decoding sequential algorithms. Moreover this approach can be easily exploited to obtain parallel algorithms: in Section 3.4 we will show how to do this for the EREW PRAM parallel model. Namely, we associate each tree node with a pair of integer numbers according to criteria dependent on the specific code. Then we sort nodes using such pairs as keys; the lexicographic order is obtained with integer (radix) sorting [35]. The obtained ordering corresponds to the order in which nodes are deleted from the tree and can thus be used to compute the code. We remark that in [74] the idea of sorting pairs has been used to obtain an *ad-hoc* linear time algorithm for the Second Neville code. In the rest of this section we show how different pair choices yield Prüfer, Neville, and Deo and Micikevičius codes, respectively.

Code	Pair $(\, x_v,\, y_v \,)$
Prüfer	$(\, \mu(v),\, d(\mu(v), v) \,)$
Second Neville	$(\, l(v),\, v \,)$
Third Neville	$(\, \lambda(v),\, d(\lambda(v), v) \,)$
Stack-Queue	$(\, l(v),\, \gamma(v) \,)$

Table 3.1: Pair (x_v, y_v) associated to node v for different codes.

3.2.1 Coding by Sorting Pairs

Let T be a rooted Cayley tree with n nodes, and let u and v be any two nodes of tree T. Let us call:

- $d(u, v)$: distance between two nodes u and v in T, $d(u, u) = 0$;
- $l(v)$: the (bottom-up) level of node v, i.e., the maximum distance of v from a leaf in T_v;
- $\mu(v)$: the maximum label among all nodes in T_v;
- $\lambda(v)$: the maximum label among all leaves in T_v;
- $\gamma(v)$: the maximum label among the leaves in T_v that have maximum distance from v;
- (x_v, y_v): a pair associated to node v according to the specific code as shown in Table 3.1;
- P: the set of pairs (x_v, y_v) for each v in T.

The following lemma establishes a correspondence between the set P of pairs and the order in which nodes are deleted from the tree.

Lemma 3.1. *For each code, the lexicographic ordering of the pairs (x_v, y_v) in set P corresponds to the order in which nodes are deleted from tree T according to the code definition.*

Proof. We discuss each code separately:

Prüfer code: notice that before a node v can be selected as a leaf the entire subtree T_v should have been deleted. Furthermore, according to the definition of Prüfer code, when the node $\mu(v)$ is chosen for deletion, T_v consists of a *chain* from $\mu(v)$ to v. All the nodes in such a chain have label smaller than $\mu(v)$ and thus will be chosen in the steps immediately following the deletion of $\mu(v)$. The tree is therefore partitioned into chains containing nodes with the same μ value and the rank of each node v in the chain is $d(v, \mu(v))$. Prüfer code deletes all the chains, in increasing order, with respect to $\mu(v)$.

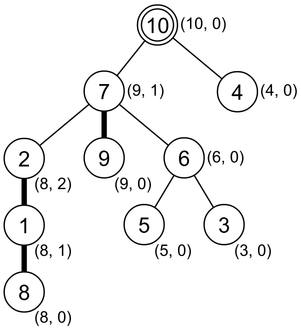
Second Neville code: the code deletes at each iteration all the leaves of T, and thus nodes are deleted starting from smaller to higher levels. Nodes within the same level are deleted in increasing label order. Hence the pair choice.

Third Neville code: it is sufficient to use the definition of pending chain given in Section 2.2.2 and to observe that, for each node v, $\lambda(v)$ is the head of the unique pending chain containing v.

Stack-Queue code: similarly to the Second Neville code, this code deletes nodes from smaller to higher levels. As proved in [37], nodes within the same level ℓ are deleted in increasing order of their γ values. The proof given by Deo and Micikevičius is by induction on ℓ.

Nodes within level 0 (i.e., the leaves of T) are such that $\gamma(v) = v$ and are deleted by increasing label order. Let u and v be two arbitrary nodes at level ℓ. According to the code definition, the order in which u and v become leaves is strictly related to the deletion order of nodes at level $\ell - 1$. Let u' and v' be the last deleted nodes of T_u and T_v respectively. It is easy to see that $l(u') = l(v') = \ell - 1$. Furthermore, by definition of γ, it holds $\gamma(u') = \gamma(u)$ and $\gamma(v') = \gamma(v)$. Since by inductive hypothesis u' is deleted before v' if and only if $\gamma(u') < \gamma(v')$, the same holds for nodes u and v.

□

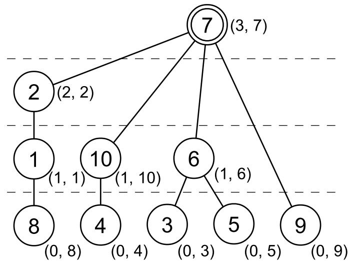


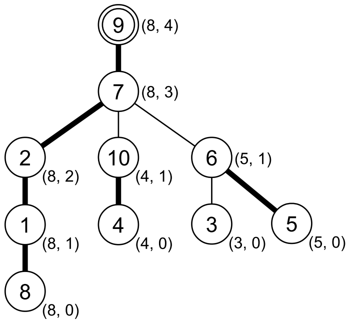
Prüfer code

Pairs:	(3,0)	(4,0)	(5,0)	(6,0)	(8,0)	(8,1)	(8,2)	(9,0)	(9,1)
Nodes:	3	4	5	6	8	1	2	9	7
Code:	6	10	6	7	1	2	7	7	10

Second Neville code

Pairs:	(0,3)	(0,4)	(0,5)	(0,8)	(0,9)	(1,1)	(1,6)	(1,10)	(2,2)
Nodes:	3	4	5	8	9	1	6	10	2
Code:	6	10	6	1	7	2	7	7	7





Third Neville code

Pairs:	(3,0)	(4,0)	(4,1)	(5,0)	(5,1)	(8,0)	(8,1)	(8,2)	(8,3)
Nodes:	3	4	10	5	6	8	1	2	7
Code:	6	10	7	6	7	1	2	7	9

Stack-Queue code

Pairs:	(0,3)	(0,4)	(0,5)	(0,8)	(0,9)	(1,4)	(1,5)	(1,8)	(2,9)
Nodes:	3	4	5	8	9	10	6	1	7
Code:	6	10	6	1	7	7	7	2	2

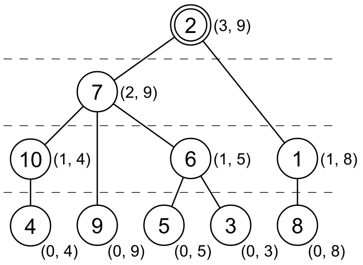


Figure 3.1: Examples of encoding Prüfer-like codes using pairs specified in Table 3.1. The pairs sorted in increasing order, the node corresponding to each pair, and the resulting codeword are shown for each code. Bold edges in the trees related to Prüfer code and Third Neville code indicate chains and pending chains, respectively; dashed lines in the trees related to Second Neville code and Stack-Queue code separate nodes at different levels.

In Figure 3.1 the pairs relative to the four codes are presented. Bold edges in the trees related to Prüfer code and Third Neville code indicate chains and pending chains, respectively; dashed lines in the trees related to Second Neville code and Stack-Queue code separate nodes at different levels. In each figure the resulting codeword, the pairs sorted in increasing order, and the node corresponding to each pair are also shown.

3.2.2 Sequential Algorithm

Our sequential coding algorithm works on rooted trees and hinges upon the pairs defined in Section 3.2.1:

```
UNIFIED ENCODING ALGORITHM
1. for each node $v$ do
2.   compute the pair $(x_v, y_v)$ according to Table 3.1
3. sort the tree nodes according to pairs $(x_v, y_v)$
4. for $i = 1$ to $n - 1$ do
5.   let $v$ be the $i$-th node in the ordering
6.   append $parent(v)$ to the code
```

Theorem 3.2. *The* UNIFIED ENCODING ALGORITHM *correctly computes Prüfer code, Second Neville code, Third Neville code, and Stack-Queue code in $O(n)$ running time.*

Proof. The correctness of the UNIFIED ENCODING ALGORITHM follows from Lemma 3.1. For all codes the information used in pairs can be easily computed in $O(n)$ time using a post-order traversal of the tree. To implement lin 2 notice that it is easy to sort the pairs (x_v, y_v) used in the encoding scheme. Indeed, independently by the specific code, each element in such pairs is in the range $[1, n]$. A radix-sort like approach [35] is thus sufficient to sort them according to y_v first and x_v later, exploiting a stable integer sorting (e.g., counting sort [35]). Then the overall running time is $O(n)$. □

We remark that the UNIFIED ENCODING ALGORITHM works on rooted trees and generates codewords of length $n-1$. According to Section 2.1.2 it can be exploited to encode unrooted trees by simply rooting them in a fixed node and omitting the last symbol of the codeword.

3.3 A Unified Decoding Algorithm

In this section we present a unified sequential algorithm for decoding Prüfer-like codes, i.e., for building the tree T corresponding to a given codeword C. As seen above, to reconstruct T, it is sufficient to compute the ordered sequence of the removed leaves, let us call it S. For each $i \in [1, n-1]$, the pair $(C[i], S[i])$ will thus be an edge in the tree ($C[i]$ and $S[i]$ represent the i-th element in C and S respectively). The decoding scheme is based on the computation of the rightmost occurrence of each value in the codeword.

3.3.1 Decoding by Rightmost Occurrence

Recall that leaves of T are exactly those nodes that do not appear in the codeword and each internal node, say v, in general may appear in C more than once; each appearance corresponds to the deletion of one of its children, and therefore implies that the degree of v decreases by 1. After the rightmost occurrence in the code, v is clearly a leaf and thus becomes a candidate for being deleted. This implies that v should appear in S after its rightmost occurrence. More formally:

$$\forall\, v \neq r, \quad \exists \text{ unique } j > rm(v, C) \ \text{ such that } S[j] = v$$

where r is the root of the tree (i.e., the last element in C) and $rm(v, C)$ denotes the index of the rightmost occurrence of node v in C. We assume that $rm(v, C) = 0$ if v does not appear in C. It is easy to compute the rightmost occurrence of each node with a simple right to left scan of C: this requires $O(n)$ time.

Code	$test(v)$	$position(v)$
Prüfer	$rm(v, C) > prev(v, C)$	$rm(v, C) + 1$
Third Neville	$rm(v, C) > 0$	$rm(v, C) + 1$
Stack-Queue	$rm(v, C) > 0$	$\|leaves(T)\| + \sigma(rm(v, C))$

Table 3.2: Condition on node v that is checked in the UNIFIED DECODING ALGORITHM and position of v as a function of $rm(v, C)$.

3.3.2 Sequential Algorithm

We now describe a decoding algorithm for Prüfer code, Third Neville code, and Stack-Queue code that is based on the rightmost occurrences. Differently from the other codes, for the Second Neville code the rightmost occurrence of each node in C gives only partial information about sequence S. Thus, we will discuss this code separately in Section 3.3.3.

We need the following notation. For each $i \in [1, n-1]$, let $\rho(i)$ be 1 if i is the rightmost occurrence of value $C[i]$, and 0 otherwise. Let $\sigma(i)$ be the number of internal nodes whose rightmost occurrence is at most i, i.e.,

$$\sigma(i) = \sum_{j \le i} \rho(j) \tag{3.1}$$

Similarly to [104], let $prev(v, C)$ denote the number of nodes with label smaller than v that become leaves before v, i.e.,

$$prev(v, C) = |\{u \, : \, u < v \text{ and } rm(u, C) < rm(v, C)\}|$$

The following lemma shows, for each code, how the position of a node in the sequence S that we want to reconstruct can be expressed in terms of rightmost occurrence of nodes.

Lemma 3.3. *Let C be a codeword of $n-1$ integers in $[1, n]$. Let* test *and* position *be defined as in Table 3.2 for Prüfer code, Third Neville code, and Stack-Queue code. Let S be the sequence of leaves deleted from the tree while* *building the codeword. The proper position in S of any node v that satisfies* $\text{test}(v)$ *is given by* $\text{position}(v)$.

Proof. We discuss each code separately, starting from the simplest one.

Third Neville code: each internal node v is deleted as soon as it becomes a leaf. Thus, the position of v in sequence S is exactly $rm(v, C) + 1$.

Prüfer code: differently from the Third Neville code, in Prüfer code an internal node v is deleted as soon as it becomes a leaf if and only if there is no leaf with label smaller than v. In order to test this condition we use information given by $prev(v, C)$: the position of v in S is $rm(v, C) + 1$ if and only if $rm(v,\ C) \geq prev(v, C)$.

Stack-Queue code: by the definition of this code, all the leaves of T, sorted by increasing labels, are at the beginning of sequence S. Then, all the internal nodes appear in the order in which they become leaves, i.e., sorted by increasing *rightmost*. Thus, the position of an internal node v is given by $|leaves(T)| + \sigma(rm(v, C))$. □

We remark that some entries of S may be still empty after positioning nodes according to Lemma 3.3. The definitions of the various codes imply that all the nodes not positioned by Lemma 3.3, except for the root, should be assigned to the empty entries of S in increasing label order. In particular, for Third Neville code and Stack-Queue code, only the leaves of T are not positioned and, in the case of Stack-Queue code, all of them will appear at the beginning of S. We are now ready to describe our UNIFIED DECODING ALGORITHM:

UNIFIED DECODING ALGORITHM

```
for each node $v$ do
  compute $rm(v, C)$
for each node $v$ except for the root do
  if $test(v)$ = true then $S[position(v)] \leftarrow v$
let $L$ be the ordered list of unused (non-root) nodes in $S$
```

	1	2	3	4	5	6	7	8	9
C:	6	10	6	7	1	2	7	7	10

	1	2	3	4	5	6	7	8	9	10
rm:	5	6	0	0	0	3	8	0	0	9
prev:	0	1	0	0	0	3	6	0	0	9
test:	t	t	f	f	f	f	t	f	f	f

	1	2	3	4	5	6	7	8	9
S (after line 3):	-	-	-	-	-	1	2	-	7
S (after line 7):	3	4	5	6	8	1	2	9	7

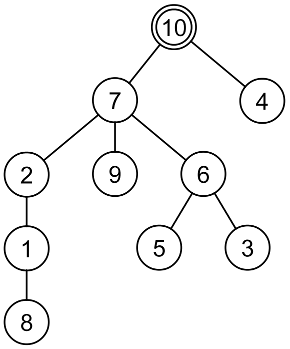


Figure 3.2: An example of execution of the UNIFIED DECODING ALGORITHM in the case of Prüfer code: content of the main data structures and tree returned as output.

```
6. let $P$ be the list of empty positions in $S$
7. for each $i = 1$ to |L| do
8.     $S[P[i]] \leftarrow L[i]$
```

where $test(v)$ and $position(v)$ are specified in Table 3.2. An example of execution of the UNIFIED DECODING ALGORITHM in the case of Prüfer code is shown in Figure 3.2.

For Third Neville and Stack-Queue code the UNIFIED DECODING ALGORITHM requires linear time, while a straightforward implementation for Prüfer code would require $O(n \log n)$ time due to the computation of *prev*. This can be reduced to $O(n)$ time by adapting the UNIFIED DECODING ALGORITHM in such a way that the *prev* computation can be avoided. Namely, lines 2–3 can be omitted (considering the $test(v)$ as `false` for each node v), and lines 6–7 can be replaced as follows:

```
6. for each $i = 1$ to |L| do
7.     $position \leftarrow \max\{first\_empty\_pos(S), rm(L[i], C) + 1\}$
8.     $S[position] \leftarrow L[i]$
```

where $first\_empty\_pos(S)$ returns the smallest empty position in S. In this implementation, nodes are considered in increasing label order: node v is assigned to position $rm(v) + 1$ of S if this position is still empty, and to the leftmost empty position otherwise. In order to see that this is equivalent to the UNIFIED DECODING ALGORITHM, observe that nodes for which $rm(v) > prev(v)$ (see the test in line 3) will always find the position $rm(v) + 1$ empty, due to the definition of $prev$. Hence, they will be inserted in S exactly as in line 3 of the UNIFIED DECODING ALGORITHM.

The performances of the UNIFIED DECODING ALGORITHM are summarized by the following theorem.

Theorem 3.4. *The* UNIFIED DECODING ALGORITHM *computes the tree corresponding to a codeword C in $O(n)$ sequential time.*

3.3.3 Second Neville Code

As we said, for this code the rightmost occurrence of each node in the codeword C gives only partial information about sequence S. Here we show how to efficiently extract from the codeword enough information to correctly decode C according with the Second Neville code. We remark that the problem of finding an optimal sequential decoding algorithm for this code was open, and our work close it.

We first observe that if all nodes were assigned with a level, sort nodes according to pairs $(l(v), v)$, as done by the encoding algorithm, would produce the sequence S (see Section 3.2.1). We now show how to compute $l(v)$ from C.

Let x be the number of leaves of T: these nodes have both level and rm equal to 0. Consider the first x elements of code C, say $C[1], C[2], \ldots, C[x]$. For each i, $1 \leq i \leq x$, such that i is the rightmost occurrence of $C[i]$, we know that node $C[i]$ has level 1. The same reasoning can be applied to get level-2 nodes from level-1 nodes, and so on. With respect to the running time, a sequential scan of code C is sufficient to compute the level of each node in linear time.

3.4 Unified Parallel Algorithms

In this section we present a parallel version of the UNIFIED ENCODING ALGORITHM proposed in Section 3.2 and of the UNIFIED DECODING ALGORITHM proposed in Section 3.3. Our algorithms are described for the classical EREW PRAM model and costs are expressed as the number of elementary operations needed to perform a task.

We chosen the PRAM theoretical model because we do not need to address any specific hardware. In the last decade, PRAM model has been deemed useless by many researchers because it is too abstract compared with actual parallel architectures. It is worth noticing that this trend is changing. At SPAA'07, Vishkin and Wen reported about the recent advancements achieved at the University of Meryland within the project PRAM-On-Chip [105]. The XMT (eXplicit Multi-Threading) general-purpose computer architecture is a promising parallel algorithmic architecture to implement PRAM algorithms. They also developed a single-instruction multiple-data (SIMD) multi-thread extension of C language with the intent of provide an easy programing tool to implement PRAM algorithms. I has primitives like: Prefix Sum, Join, Fetch and Increment, etc.

An optimal PRAM algorithm for encoding Prüfer codes, which improves over a previous result due to Greenlaw and Petreschi [54], is given in [53]. A few simple changes make the algorithm works also for the third Neville code. In [27] non optimal encoding algorithm for Prüfer has been presented, it makes use of the idea of sorting pairs but requires $O(n \log n)$ operations. Efficient, but not optimal, parallel encoding algorithms for the Second Neville code and the Stack-Queue code have been presented in [37]. Our unified algorithm either matches or improves by a factor $O(\sqrt{\log n})$ the performances of the best *ad-hoc* approaches known so far.

Concerning decoding, Wang, Chen, and Liu [104] propose an $O(\log n)$ time decoding algorithm for Prüfer code using $O(n)$ processors on an EREW PRAM. To the best of our knowledge, parallel decoding algorithms for the other Prüfer-like codes were not known in the literature until our work. Namely, we designed the first parallel decoding algorithm for Second Neville

Code	**Encoding**		**Decoding**	
	known	*our result*	*known*	*our result*
Prüfer	$O(n)$[53]	$O(n)$	$O(n \log n)$[104]	$O(n \log n)$
Second Neville	$O(n \log n)$[37]	$O(n\sqrt{\log n})$	open	$O(n\sqrt{\log n})$
Third Neville	$O(n)$[37, 53]	$O(n)$	open	$O(n\sqrt{\log n})$
Stack-Queue	$O(n \log n)$[37]	$O(n\sqrt{\log n})$	open	$O(n\sqrt{\log n})$

Table 3.3: Summary of our results on the EREW PRAM model. Costs are expressed in terms of number of operations.

code, Third Neville code, and Stack-Queue code: our unified algorithm works on a n-processors EREW PRAM in $O(\log n)$ time with cost $O(n\sqrt{\log n})$. For Prüfer code, the cost of our algorithm is $O(n \log n)$ and matches the best previous result [104]. Our parallel results both for encoding and for decoding are summarized in Table 3.3.

3.4.1 Encoding

Before showing how to parallelize each step of the UNIFIED ENCODING ALGORITHM, we want to remark that if the tree is unrooted, the Euler tour technique makes it possible to root it in $O(\log n)$ time with cost $O(n)$ [59].

We now discuss how to compute all information that constitutes the pair components as described in Section 3.2.

Lemma 3.5. *The pairs given in Table 3.1 can be computed on the EREW PRAM model in $O(\log n)$ time with cost $O(n)$.*

Proof. We discuss separately the components of each pair.

$\mu(v)$: the maximum node in each subtree can be computed in $O(\log n)$ time with cost $O(n)$ using the Rake technique [59] as done in [54]. In order to avoid concurrent reading during the Rake operation, the tree T must be preliminarily transformed into a binary tree T_R as follows: for each node v with $k > 2$ children, v is replaced by a complete binary tree of height $\lceil \log k \rceil$ having v as root and v's children as the k leftmost leaves. This transformation can be also done in $O(\log n)$ time with cost $O(n)$ [53].

$d(\mu(v), v)$: we partition T into chains by marking each node v with the value $\mu(v)$ and by deleting edges between nodes with different μ values. Now, the rank of node v in its chain is exactly $d(\mu(v), v)$. In order to compute the chains, each node links itself to its parent if $\mu(v) = \mu(parent(v))$. A List Ranking then gives the position of each node in its chain in $O(\log n)$ time with cost $O(n)$ [59]. The use of the binary tree T_R guarantees that no concurrent read is necessary for accessing $\mu(parent(v))$.

$l(v)$: an Euler tour gives the distance $d(v, r)$ of each node v from the root r of tree T. Then, $l(v) = d(f, r) - d(v, r)$, where f is a leaf of T_v at maximum distance from r: f can be easily computed using the Rake technique [59].

$\lambda(v)$: the same techniques used for computing $\mu(v)$ can be adapted to obtain the maximum leaf of each subtree with the same performances.

$d(\lambda(v), v)$: analogous considerations as for computing $d(\mu(v), v)$ hold.

$\gamma(v)$: given the distance of each node from the root, $\gamma(v)$ is the node $u \in T_v$ such that $(d(u, r), u)$ is maximum and can be computed with the Rake technique.

□

The following theorem summarizes the performances of the UNIFIED ENCODING ALGORITHM in a parallel setting.

Theorem 3.6. *On the EREW PRAM model, the* UNIFIED ENCODING ALGORITHM *computes Prüfer code and Third Neville code optimally (i.e., in $O(\log n)$ time with cost $O(n)$) and Second Neville code and Stack-Queue code in $O(\log n)$ time with cost $O(n\sqrt{\log n})$.*

Proof. By Lemma 3.5, line 1 of the UNIFIED ENCODING ALGORITHM requires $O(\log n)$ time with cost $O(n)$. Line 3 can be trivially implemented in $O(1)$ time with cost $O(n)$. The sorting in line 2 is thus the most expensive operation. Following a radix-sort like approach and using the stable integer-sorting algorithm presented in [55] as a subroutine, line 2 would require $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on an EREW PRAM[1]. This gives the stated running time and cost for Second Neville code and Stack-Queue code. For Prüfer code and Third Neville code we can further reduce the cost of our algorithm to $O(n)$ by using a more clever sorting procedure that benefits from the partitioning of the tree into chains.

Let us consider Prüfer code first. As observed in [54], the final node ordering can be obtained by sorting chains among each other and nodes within each chain. In our framework, the chain ordering is given by the value $\mu(v)$, and the position of each node within its chain by the distance $d(\mu(v), v)$. Instead of using a black-box integer sorting procedure, we exploit the fact that we can compute optimally the size of each chain, i.e., the number of nodes with the same $\mu(v)$, by means of prefix sums. Another prefix sum computation can then be used to obtain, for each chain head, the overall number of nodes in the preceding chains, i.e., its final position. At last, the position of the remaining nodes is univocally determined by summing up the position of the chain head $\mu(v)$ with the value $d(\mu(v), v)$. Similar considerations can be applied to the Third Neville code. □

We remark that our algorithm solves within a unified framework the parallel encoding problem. With respect to Prüfer code and Third Neville code, it matches the performances of the (optimal) algorithms known so far [37, 53]. With respect to Second Neville code and Stack-Queue code, it improves of an $O(\sqrt{\log n})$ factor over the best approaches known in the literature [37].

[1] The result on parallel integer sorting in [55] holds when the machine word length is $O(\log n)$. Under the more restrictive hypothesis that the word length is $O(\log^2 n)$, the cost of sorting can be reduced to $O(n)$, and so does the cost of our algorithm.

3.4.2 Decoding

We now show how to parallelize the UNIFIED DECODING ALGORITHM; as done in Section 3.3 we consider the Second Neville code separately.

The following lemma analyzes the rightmost computation in parallel.

Lemma 3.7. *The rightmost occurrences of nodes in a codeword C of length $n-1$ can be computed in $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on the EREW PRAM model.*

Proof. We reduce the rightmost occurrence computation to a pair sorting problem: we sort in increasing order the pairs $(C[i], i)$, for $i \in [1, n-1]$. Indeed, in each sub-sequence of pairs with the same first element $C[i]$, the second element of the last pair is the index of the rightmost occurrence of node $C[i]$ in the code. Since each pair value is an integer in $[1, n]$, we can again use twice the stable integer-sorting algorithm described in [55]: this requires $O(\log n)$ time and $O(n\sqrt{\log n})$ cost on an EREW PRAM. Then, each processor p_i in parallel compares the first element of the i-th pair in the sorted sequence to the first element of the $(i+1)$-th pair, determining whether i corresponds to the end of a subsequence or not. This requires additional $O(1)$ time and linear cost with exclusive read and exclusive write operations. □

The performances of the UNIFIED DECODING ALGORITHM in a parallel setting are described by the following theorem.

Theorem 3.8. *The* UNIFIED DECODING ALGORITHM *computes the tree corresponding to a codeword C, on the EREW PRAM model, in $O(\log n)$ time with cost $O(n \log n)$ for Prüfer code and $O(n\sqrt{\log n})$ for Third Neville code and Stack-Queue code.*

Proof. With respect to Prüfer code, the parallel version of the UNIFIED DECODING ALGORITHM yields essentially the same algorithm described in [104]. Its bottleneck is the *prev* computation that can be reduced to a dominance counting problem and can be solved on the EREW PRAM in $O(\log n)$ time with cost $O(n \log n)$ [5, 32]: we refer to [53, 104] for a detailed analysis. For the other codes, $\sigma(i)$ (defined in Equation 3.1) can be computed for each i using a prefix sum operation [59]. In order to compute list L in line 4, we can mark each node not yet assigned to S and obtain its rank in L by computing prefix sums. Similarly for list P in line 5. Hence, the most expensive step is the rightmost computation, which requires integer sorting (Lemma 3.7). □

Second Neville code

Unfortunately, the approach used in Section 3.3.3 is inherently sequential and thus inefficient in parallel. We now discuss an alternative approach for computing the level of each node v, from a codeword C. This approach can be easily parallelized.

Lemma 3.9. *Let C be the Second Neville code codeword for a tree T. The level of each node in T can be computed from C on the EREW PRAM model in $O(\log n)$ time with cost $O(n\sqrt{\log n})$.*

Proof. Let T' be the tree obtained by decoding C with the Stack-Queue code: although T and T' are different, the level of each node is the same both in T and T'. Indeed, as shown in Table 3.1, for both Second Neville code and Stack-Queue code the first element of the pair is $x_v = l(v)$.

Then, after T' is build using the UNIFIED DECODING ALGORITHM, we compute node levels applying the Euler tour technique on it. We remark that the Euler tour technique requires a particular data structure [59] that can be built as described in [54]. The bottleneck of this procedure is sorting pairs of integers in $[1, n]$ and thus, once again, we can exploit the parallel integer sorting presented in [55]. □

Given level information, the correct sequence S corresponding to tree T can be easily obtained by sorting the pairs $(l(v), v)$. We can summarize the results concerning the Second Neville code as follows:

Theorem 3.10. *The tree corresponding to a codeword C according to the Second Neville code can be computed in $O(n)$ sequential time and in $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on the EREW PRAM model.*

Proof. The correctness follows from the definition of the Second Neville code and from Lemma 3.1. The running time is guaranteed by Lemma 3.9 and by the bounds on integer sorting given in [55]. □

3.5 Concluding Remarks

In this chapter we have presented a unified approach for coding labeled trees by means of strings of node labels and have applied it to four well-known Prüfer-like codes due to Prüfer [90], Neville [79], and Deo and Micikevičius [39]. The encoding scheme hinges upon the definition of pairs associated to the nodes of the tree according to criteria dependent on the specific code: the coding problem is then reduced to the problem of sorting these pairs in lexicographic order. The decoding scheme is based on the computation of the rightmost occurrence of each label in the code. In particular, we obtained the first linear time sequential decoding algorithm for the Second Neville code.

We have also shown how it is possible to parallelize our unified encoding and decoding algorithms. There where no decoding algorithms for the Second Neville code, the Third Neville code, and Stack-Queue code before our work. Concerning the encoding our results either improve or match the best results known in the literature. Moreover, since integer sorting is the most expensive operation in our parallel algorithms, any improvement on the computation of integer sorting directly improves our results. The only exception is the Prüfer decoding parallel algorithm: here the dominance counting problem is the bottleneck. In [5] the lower bound $\Omega(n \log n)$ has been shown for the dominance counting problem in parallel. To the best of our knowledge, it is an open question to understand if it is possible to overcome this bound when the input is limited to n integer values in the range $[1, n]$ or to avoid the *prev* computation in the decoding algorithm for Prüfer code.

Algorithmic aspects related with encoding and decoding Prüfer-like codes have been analyzed also in [38] where a different scheme relaying on lists has been presented. Codes have been classified in function of used lists (FIFO, LIFO, or sorted lists), the ordering of initial leaves, and whether they require a single list or multiple lists.

Chapter 4

Applications of Tree Encodings

Labeled trees are of interest in theoretical and practical areas of computer science. They are used in a great variety of applications ranging from Phylogenetic Trees to data compression, from XML data representation and indexing to the computation of graph volumes. Tree encodings are used in applications like Fault Dictionary Storage [12], Distributed Spanning Tree Maintenance [48], etc.

In this chapter we focus on two applications of tree encodings: random trees generation and Genetic Algorithms. The first application shows how these combinatorial bijection can be fruitfully exploited to guarantee that the trees are generated uniformly at random, both in sequential and parallel settings. Genetic Algorithms provide a wider example of application of tree encodings. In this context the choice of an appropriate representation for trees is fundamental. Many experimental comparisons have been presented in the literature in order to explore several possible tree encodings. Some of these experiments driven our attention on certain bijective code not belonging to the class of Prüfer-like code. These codes will be studied in Chapter 5.

This chapter is organized as follows: Section 4.1 describes the use of Prüfer-like codes to generate trees uniformly at random, we also show experimental results proving that this method is competitive with other fast methods to generate random trees. In Section 4.2 we give a brief introduction to Genetic Algorithms and discuss known experimental results on trees.

4.1 Generating Random Trees

The problem of generating a random tree with certain properties is fundamental in Computer Science, especially in order to run experiments and simulations. Its has been widely studied both in a sequential setting (see for example [40, 70]) and in a parallel setting (see for example [36]). Roughly speaking, this task can be performed in several ways: for example, by adding random edges until the graph is completely connected, then breaking cycles without disconnecting the graph. Another approach is the following: construct the tree by adding nodes at random, connecting them to nodes already in the tree. Other methods are possible. Easy methods often require more than $O(n)$ time, while efficient ones may have the drawback that the random choice is not uniformly distributed among the set of all possible trees. As shown by the following example, adding nodes at random, the star of n nodes can be generated with a probability considerably higher than any given n-nodes path.

Example 9. We want to generate random rooted Cayley trees by adding nodes at random. At each step we select a random unused label v and a random used one p an we add node v to the tree as a child of p. The first chosen node will be the root. Now consider the probability that the star rooted at node 1 is generated, call it S_1. At the first step we must chose label 1, the probability that it happen is $Pr^{(1)}[v = 1] = \frac{1}{n}$. In the following steps we can choose any label v but as p we have to choose always 1; the probability that it happen depends on how many nodes are already been added to the tree. At the second step $Pr^{(2)}[p = 1]$ (the probability that we choose $p = 1$) is 1, at the third step $Pr^{(3)}[p = 1] = \frac{1}{2}$, at the fourth step $Pr^{(4)}[p = 1] = \frac{1}{3}$, an so on. In general at step i we have $Pr^{(i)}[p = 1] = \frac{1}{i-1}$. The overall probability for S_1 to be generated is:

$$Pr[S_1] = \frac{1}{n} \prod_{i=2}^{n} \frac{1}{i-1} = \frac{1}{n!}$$

Let us now compute the probability that a given path P is generated; w.l.o.g. we assume that P is rooted at node 1 and the node sequence is $1, 2, 3, \ldots, n$ (any other path has the same probability). At the first step we have to choose

$v = 1$ as root: $Pr^{(1)}[v = 1] = \frac{1}{n}$. At the second step we must choose $v = 2$ and $p = 1$: $Pr^{(2)}[v = 2] = \frac{1}{n-1}$ and $Pr^{(2)}[p = 1] = 1$. At the third step we must choose $v = 3$ and $p = 1$: $Pr^{(3)}[v = 3] = \frac{1}{n-2}$ and $Pr^{(3)}[p = 2] = \frac{1}{2}$. In general at step i we have $Pr^{(i)}[v = i] = \frac{1}{n-i+1}$ and $Pr^{(i)}[p = i - 1] = \frac{1}{i-1}$. The overall probability for given path P to be generated is:

$$Pr[P] = \frac{1}{n} \prod_{i=2}^{n} \frac{1}{n-i+1} \prod_{i=2}^{n} \frac{1}{i-1} = \frac{1}{n!} \frac{1}{(n-1)!}$$

Both S_1 and P are Cayley trees in $\mathcal{R}_n$ but $Pr[S_1]$ is considerably higher than $Pr[P]$.

Thus generate Cayley trees by adding nodes at random does not guarantee uniform distribution (even for trees that are not stars nor paths).

Generating a random codeword of $n - 1$ integers in the range $[1, n]$ and applying a decoding algorithm is an easy and fast way to generate a random tree. It also guarantees that, if each integer is chosen uniformly at random in $[1, n]$, each rooted Cayley tree will have the same probability to be generated. Moreover, we have experimentally verified that this method is competitive, in terms of running time, with the one based on random leaves addition.

4.1.1 Experimental Comparison

In order to verify if the generation of random trees based on Prüfer-like code has good performances, we compared it with an effective implementation of the add-leaves-at-random idea used in Example 9. In order to select, at each step, a random leaf to add we precompute a permutation of $[1, n]$. It is well known that the following algorithm ensures that the permutation is chosen uniformly at random:

```
for $i = 1$ to $n$ do
    $perm[i] = i$
for $i = 1$ to $n$ do
    swap $perm[i]$ and $perm$[Random($i$,$n$)]
```

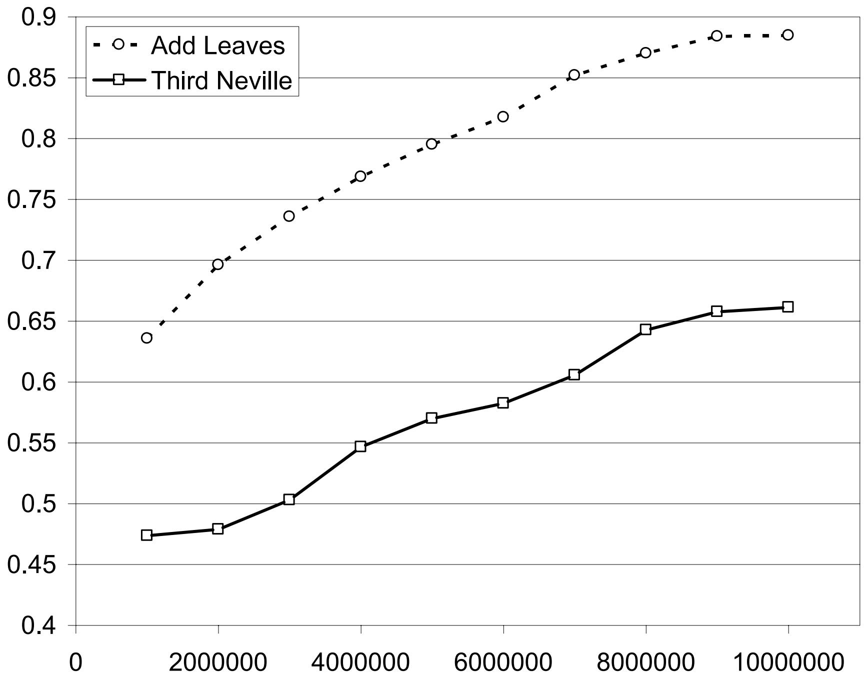


Figure 4.1: Experimental comparison of Random Tree Generation algorithms, from 1 to 10 million nodes. Y-axis report the time required to generate a single tree in milliseconds. Decoding a random codeword with Third Neville code is 30% faster than the Add Leaves based method.

The function `Random(`i`,`n`)` returns a random value in $[i, n]$. Then nodes are added to the tree in the order given by vector `perm`. Each node $v = perm[i]$ is attached to one of those already added to the tree, i.e., all nodes between $perm[1]$ and $perm[i-1]$.

```
1. $T = ([1, n], \varnothing)$
2. for $i = 1$ to $n$ do
3.     add the edge $(perm[i], perm[$Random(1,$i-1$)$])$
```

This algorithm has been compared against the decoding of a random codeword. We choose the Third Neville code implementing of the UNIFIED DECODING ALGORITHM presented in Section 3.3.

Both algorithms have been implemented in standard ansi C (C99 revised standard), compiled with gcc (version 3.3.5) with optimization flag `O3`.

Random values have been produced by the `rand()` pseudo-random source of numbers provided by the ANSI C standard library. We used only odd seeds to initialize the random generators and we randomly generated the sequence of seeds used in each test starting from a base seed. Trees are implemented through adjacency lists. Our experiments have been carried out on a workstation equipped with two Dual Core Opteron processors with 2.2 GHz clock rate, 6 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Debian (kernel 2.6.8). The running time of each experiment has been measured by means of the standard system call `getrusage()`.

Trees from 1 million to 10 million nodes have been generated. Results (reported in Figure 4.1) clearly show that decoding a random codeword with Third Neville code is 30% faster than the Add Leaves based method. Thus we conclude that using tree encodings is definitely the best way to generate random trees.

4.1.2 Constrained Random Trees

Using Prüfer-like codes certain topological properties of a tree are explicit in its codeword. This it is possible to impose constrains on the random trees generated maintaining linear running time.

Let C be the codeword for a tree T obtained with any Prüfer-like code: since each node appears in C a number of time equal to the number of its children, information concerning degrees, leaves, and root are explicit in C (see Property 2.4). Thus we can state that:

- to generate a tree rooted in a desired node r it is enough to ensure that $C[n-1] = r$;
- to let a certain node v be a leaf, it does not have to appear in C at all;
- if the set of leaves must be $\{v_1, v_2, \ldots, v_k\}$, all nodes but these must appear in C at least once;

- to guarantee that a node v gets degree d in T, it must appear exactly $d-1$ times in C (d times if v is the root).

All these observations can be combined together to generate random trees satisfying several constraints. This same reasoning can be exploited to generate unrooted trees by decoding $n-2$ length random codewords.

4.1.3 Parallel Random Trees Generation

Generating random trees in a parallel setting is an hard task. Add nodes at random is an inherently sequential method, a straightforward use of this idea would result in a misuse of the underling parallel architecture. On the other hand, if each processor attaches a node to some other random node, disregarding other processors activity, there is no guarantee that the resulting graph is connected and acyclic; repair such a graph to obtain a tree is expensive.

The idea of using bijective code to generate random tree is the easiest one. Indeed in order to generate a codeword each processors can choose a random number independently, then the tree is obtained directly with a parallel decoding algorithm (see Section 3.4). This idea has been exploited in [36] where a modified Prüfer code was used to obtain an architecture-specific (8192-processors MAS-Par MP-1) almost-constant-time algorithm to generate random trees. This result hinges upon the fact that, when $n < 8192$, they have an almost-constant-time integer sorting algorithm for MAS-Par MP-1. The algorithms shown in Section 3.4 provides an architecture-independent solution for this problem. Random trees can be generated taking advantage from the best integer sorting algorithm available on each specific architecture.

4.2 Genetic Algorithms

Genetic Algorithms (GAs) are search heuristics that hinge upon the evolutionary ideas of natural selection and genetic. The basic concept of GAs is to simulate natural processes necessary for evolution, specifically those that follow the principles of survival of the fittest. They represent an intelligent exploitation of a random search within a defined search space to solve a problem. We will focus on GA whose search space is the set of trees, e.g., GA for finding the Minimum Spanning Tree of a graph with additional constrains: minimum diameter, fixed number of leaves, bounded maximum degree, etc. (see for example [41, 106].)

A GA starts with a *population* of a certain number of random candidate solution, called *individuals*, that in our case are simply random trees. Each individual is represented by its *chromosome*: a code (usually a string) that identifies the individual. Individual are compared according to a fitness function and a set of good ones is selected. The fitness function strictly depends on the problem we are dealing with, for example if we are looking for MST with minimum diameter meaning full fitness functions should assign higher fitness to trees with smaller diameter. There are many criteria to determine how to choose good individuals given their fitness, a deep discussion of them is outside the scope of this thesis (we refer the interested reader to [75, 92]). We just mention that the easiest criterion is to determine whether to keep or discard an individual at random using a probability proportional to its fitness.

Then selected individuals are used to produce offsprings via genetic operators, in this way a new *generation* of individual is obtained. Several genetic operators have been introduced in the literature along the years, the most used are *crossover* and *mutation*. In a crossover, two chromosomes are mixed together (according to several possible criteria) to obtain a new chromosome, the underlying idea is that the offspring individual should inherit parent's peculiarity and then it has chances to be better than both. In a mutation, the chromosome of a single individual is slightly changed (at random), this may or may not improve its fitness.

The GA repeats this selection/reproduction scheme until one among many possible stop-criteria is reached:

- a solution "good enough" is generated;
- the improvement in the fitness of the best individual is negligible with respect to the previous generations;
- the number of generations has reached a given bound;
- the computational time has reached a given bound.

An effective individual representation and meaningful fitness evaluation are the keys of the success in GAs. We refer the interested reader to [75, 92] for a more detailed description of the fundaments of GAs.

While the fitness function strictly depends on the specific problem we are trying to solve, the individual representation by means of chromosome strings only depends on the solution space. There exist several tree representations suitable for GA, but not all of them achieve good results. There are certain desirable properties for a code in GA. It should:

be injective: it should be able to represent each tree with a different codeword;

be unbiased: each tree should be represented by the same number of codewords;

be surjective: every codeword should represent a tree;

have high locality: small changes in the tree should correspond to small changes in the codeword, and vice versa;

have high heritability: when a codeword is obtained by mixing two codewords (ancestors) each edge of the offspring tree should belong to either of the ancestor trees;

be efficient: the encoding and decoding, should require small running time, in order to efficiently compute the fitness.

Among the others [82, 96] Prüfer-like code look like an appealing choice because they are bijective and linear time encoding and decoding algorithms

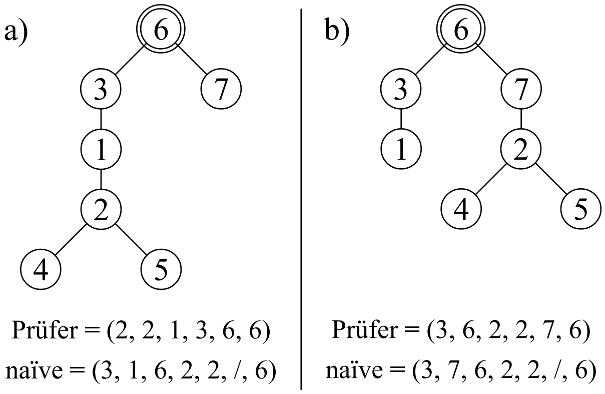

a)
6
3
7
1
2
4
5
Prüfer = (2, 2, 1, 3, 6, 6)
naïve = (3, 1, 6, 2, 2, /, 6)
b)
6
3
7
1
2
4
5
Prüfer = (3, 6, 2, 2, 7, 6)
naïve = (3, 7, 6, 2, 2, /, 6)

Figure 4.2: a) A tree T, and corresponding Prüfer code and naïve code. b) T' obtained form T changing edge $(2, 1)$ in $(2, 7)$, together with corresponding Prüfer code and naïve code.

are known. Unfortunately it has been experimentally observed that Prüfer-like codes preform poorly in GA because they have poor locality and heritability [51]. In all Prüfer-like codes the tree topology determines the elimination order of nodes, so a small change in the tree may cause a variation of this order and thus a big change in the string (see Figure 4.2a and 4.2b). This is the reason why Prüfer and Prüfer-like codes exhibit low locality and heritability [51].

In order to better understand how a code can exhibit high locality and heritability we now consider the *naïve code*. This code represents a rooted tree simply listing the parent of each node (see Figure 4.2a). Each edge $(v, p(v))$ of a tree corresponds to the v-th element of the codeword, thus this code has maximal locality: a single change in the tree corresponds to a single change in its codeword, and vice versa (see 4.2b). Naïve code also has high heritability. Consider two trees T_1 and T_2 and their codewords C_1 and C_2. Let C be a string obtained by mixing C_1 and C_2 with a crossover and T the corresponding tree. Since, for each i, either $C[i] = C_1[i]$ or $C[i] = C_2[i]$ we deduce that each edge $(v, p(v))$ in T either comes from T_1 or from T_2. Notice that T_1 and T_2 must have the same root in order to avoid that C contains 0 or 2 entries identifying a root, for this purpose it is enough to re-root T_2 in the root of T_1 (or vice versa) before the crossover.

Unfortunately, naïve code is not bijective, certain codewords may represent graphs not necessarily connected or containing cycles or loops. This implies that codewords obtained by crossover or mutation are not necessarily trees: more precisely, the probability of obtaining a tree is $\frac{1}{n}$. This is a serious shortcoming for naïve code to be used in GAs.

In [61], an experimental analysis shows that locality and heritability properties are satisfied by Blob code much better than by the Prüfer code. The Blob code is a bijective code introduced by Picciotto [89]. Like Prüfer-like codes, all of them are bijection between Cayley trees and codewords but they do not belong to the class of Prüfer-like codes because they are not base on recursive leave elimination. Blob code has also been shown to be competitive against other well known tree representations in GAs [61, 62, 91].

Unfortunately these promising experimental results do not provide any insight on the underlying reasons that make the Blob code better than the Prüfer code in this field. For this reason we decide to study the three codes presented by Picciotto: Blob code, Happy code, and Dandelion code. In the next chapter, a deep discussion of these code is presented. Here it is enough to say that our study clarified the reasons why Blob code has good locality and heritability. Moreover, interpreting the three codes as transformation of trees into functional digraphs, we pointed out that the Dandelion code approximate the desirable properties held by the naïve code much better than the Blob code. So, in a paper published in 2005 [25], we conjectured that in GA Dandelion code (and a modified version of the Happy code we introduced) should outperform Blob code. This assertion has been experimentally verified in 2006 by Paulden and Smith [84, 85, 101]: moreover the proved that our Modified Happy code has similar locality properties and slightly better heritability properties to the Dandelion code [85].

4.3 Concluding Remarks

In this chapter we have seen two among many possible applications of tree encoding. We have shown that they lead to efficient and unbiased algorithms to generate random trees, both in sequential and parallel settings.

In the field of Genetic Algorithms, where the choice of a good encoding play a crucial role, experimental analysis shown that Prüfer-like codes are outperformed by other bijective codes. In order to understand the underlying reasons behind these results we decided to better investigate other codes. In the next chapter a deep discission about these codes is presented. Our study brought us to the definition of a general encoding and decoding scheme based on the transformation of a tree into a functional digraph. This makes it possible for us to obtain linear time algorithm for encoding and decoding all Picciotto's codes.

Chapter 5

Transformation Codes

In Chapters 2 we have shown how Prüfer-like codes can be encoded and decoded in optimal linear time. In Chapters 4 however we reported that they lack other desirable properties. As observed in [51], Prüfer codes are a poor tree representation for Genetic Algorithms, since they do not have good locality and heritability (see Section 4.2). Experimental analysis [61] shown that these properties are much better satisfied by the Blob code described by Picciotto in her PhD thesis [89].

These experimental results do not provide any insight on the underlying reasons that make the Blob code better than the Prüfer code in this field. So our interest has been stimulated and therefore we decide to study, from an algorithmic point of view, all the three codes described by Picciotto in [89]: Blob code, Happy code and Dandelion code.

This chapter is organized as follows: initially we recall the original algorithms given by Picciotto for her codes. We also recall the E-R Bijection: a code introduced by Eğecioğlu and Remmel [43] several years before Picciotto's work. In Section 5.4, as a novel result of this thesis, we present a general scheme for defining bijective codes based on the transformation of a tree into a functional digraph. We show how it is possible to map Picciotto's codes to our scheme (for this reason we call them Transformation codes). This gives us a better comprehension of how encoding preserves the topology of the tree, and therefore it helps to understand which code better fits desirable properties, such as locality and heritability [24, 25].

It should be remarked that the Dandelion code is basically equivalent to the E-R Bijection and that the work by Eğecioğlu and Remmel inspired our general scheme. We also want to highlight that the general scheme based on graph transformation introduced in this chapter is capable to describe all possible bijective tree codes, while only a strict subset of them can be described as a Prüfer-like code. In the literature other codes for Cayley tree not considered in this thesis have been introduced (e.g., Chen [29], Palmer e Kershenbaum [82]) as well as their generalizations and specializations. For example there are code designed to describe trees that are spanning trees of bipartite or multipartite graphs (e.g., Ω_n bijection [44], Rainbow code [83]). Let us now introduce a few preliminary definitions and notations.

5.1 Preliminaries

In order to keep our description coherent with the one given by Picciotto, in this chapter we will deal with unrooted Cayley trees, labeled with integers in $[0, n-1]$ rather than $[1, n]$. Moreover all trees will be considered as rooted at node 0 with edges oriented upwards, from a node to its parent.

Definition 5.1. Given a function $g : A \to A$, the *functional digraph* $G = (V, E)$ associated with g is a directed graph with $V = A$ and $E = \{(v, g(v)) : v \in V\}$.

It is well known that:

Lemma 5.2. *A digraph $G = (V, E)$ is a functional digraph if and only if the out degree of each node is equal to 1.*

Corollary 5.3. *Each connected component of a functional digraph is composed of several trees, each of which is rooted in a node belonging to the* core *of the component, which is either a cycle or a loop (see Figure 5.1a).*

Functional digraphs are easily generalizable to represent functions undefined in some point: if $g(x)$ is not defined, the node x in G does not have

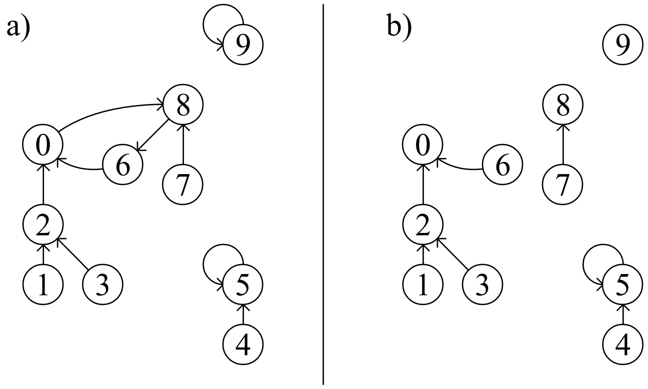


Figure 5.1: a) A functional digraph associated with a fully defined function. b) A functional digraph associated with a function undefined in 0, 8, and 9.

outgoing edges. The connected component of G containing an x, such that $g(x)$ is not defined, is a tree rooted at x without cycles (see Figure 5.1b). In the following loops will always be considered as cycles of length 1.

Remark 5.4. Let T be a rooted tree and $p(v)$ be the parent of v for each node v in T. T is the functional digraph associated with the function p.

Using the notation $pathset(u, v)$ we refer to the set of nodes in the directed path, between u and v. For our purposes we will assume that u and v do not belong to $pathset(u, v)$. As an example, consider the digraph of Figure 5.1a, $pathset(3, 6)$ is $\{0, 2, 8\}$.

5.2 Picciotto's Codes

In this section we recall Blob code, Happy code, and Dandelion code as originally presented by Picciotto in her PhD thesis [89]. As she explicitly remarks, all of them hinges upon previous works. The first one gives explicitly a bijection that in the Orlin's proof of Cayley's theorem appears in implicit form [81]. The Happy code is based on a proof by Knuth [69]. The last one is an implementation of the Joyal's pseudo-bijective proof of Cayley's theorem [60] and is equivalent to the code introduced in [43] by Eğecioğlu and Remmel: the E-R Bijection. Picciotto initially studied these codes in terms of matrix transformations then, by means of the Kirchhoff's Matrix

Tree Theorem [102], she presented them as algorithms on trees. In this thesis we focus only on algorithmic aspects related with these codes.

For all codes we recall only the encoding algorithm, since it is sufficient to map these codes into our general scheme based on digraph transformation. In this way we will obtain new linear time encoding algorithms; decoding algorithms will be provided in terms of inverse transformations.

In this section we also explicitly recall the E-R Bijection, in order to show that the Dandelion code is equivalent to this formerly introduced code.

5.2.1 Blob Code

Let us consider a tree with n nodes labeled with distinct integers in $[0, n-1]$ rooted at node 0. The encoding algorithm for the Blob code consider all nodes but 0 in decreasing label order. Each node is detached from its parent and added to a macro node called *blob*. This macro node has a parent in the tree (a conventional node) and it may contain many nodes; each node included in *blob* maintains its own subtree, if any (the example in Figure 5.2 provide a clarifying image). Nodes whose ascending path intersect the *blob*, once detached, force the *blob* to change its parent, others do not. The formers add to the codeword the parent of *blob* before the induced change, while the others simply add their parents.

Formally the encoding algorithm can be described as follows:

BLOB ENCODING ALGORITHM

```
Root $T$ in $0$
Initialize $C$ as an empty string of size $n-2$
$blob = \{n-1\}$
for $v = n-2$ down to $1$ do
  if $(pathset(v,0) \bigcap blob) \neq \varnothing$ then
    $C[v-1] = p(v)$
    delete $(v, p(v))$
    insert $v$ in $blob$
  else
```

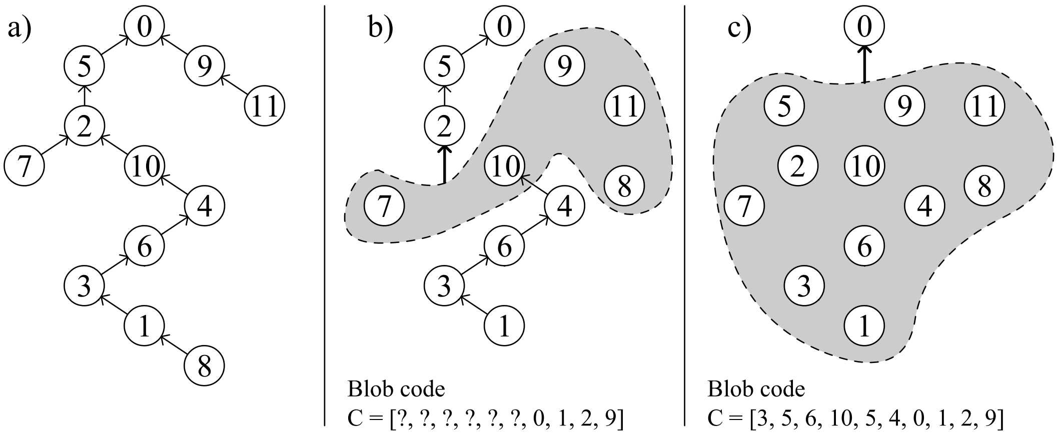

a)
0
5
9
11
2
7
10
4
6
3
1
8
b)
0
5
9
11
2
10
8
7
4
6
3
1
Blob code
C = [?, ?, ?, ?, ?, ?, 0, 1, 2, 9]
c)
0
5
9
11
2
10
8
7
4
6
3
1
Blob code
C = [3, 5, 6, 10, 5, 4, 0, 1, 2, 9]

Figure 5.2: a) A sample tree T rooted at node 0; b) an intermediate step of the execution of the BLOB ENCODING ALGORITHM. The grey area identify the *blob*, question marks in the codeword correspond to values that are still unassigned; c) the resulting codeword at the end of the execution.

```
10.         $C[v-1] = p(blob)$
11.         delete $(blob, p(blob))$
12.         add $(blob, p(v))$
13.         delete $(v, p(v))$
14.         insert $v$ in $blob$
```

In Figure 5.2 an example of the execution of BLOB ENCODING ALGORITHM is given.

The computational complexity of original Blob encoding and decoding algorithms is quadratic in the number of nodes of the tree, due to the computation of $pathset(v, 0)$ at each iteration (line 5).

Improvements

We now show how it is possible to improve the BLOB ENCODING ALGORITHM to obtain a linear time algorithm.

We will call *stable* all nodes satisfying the test in line 5 because they let the *blob* parent unchanged. The value in the code corresponding with a stable node v is simply $p(v)$.

Analyzing this algorithm we can see that the condition in line 5 is not strictly connected to the incremental construction of the *blob*, but it can be computed apriori as Lemma 5.5 asserts:

Lemma 5.5. *Stable nodes are all nodes v such that $v < max(pathset(v, 0))$.*

Proof. When node v is considered by the encoding algorithm the set *blob* contains all the nodes from $v + 1$ to n. Then the condition of line 5 holds if and only if at least a node greater than v occurs in $pathset(v, 0)$. Remember that $pathset(v, 0)$ does not include v and 0. □

Lemma 5.6. *For each unstable node v, let z be the smaller unstable node greater than v, $p(z)$ is the value inserted in the code when v is considered by the encoding algorithm.*

Proof. In line 10 of BLOB ENCODING ALGORITHM the current parent of *blob* defines the code value corresponding to an unstable node v. In subsequent lines the *blob* becomes child of $p(v)$. It implies that $p(blob)$ is equal to the parent of the smaller unstable node greater than v, i.e. $p(z)$. □

Our characterization of stable nodes decreases the complexity of BLOB ENCODING ALGORITHM to $O(n)$.

5.2.2 Happy Code

The encoding algorithm for the Happy code focuses on the path from 1 to 0. Since the aim of the algorithm is to ensure the existence of edge $(1, 0)$, all nodes on the original path from 1 to 0 are sequentially moved in order to form cycles. Let us call *maximal* each node v in $pathset(1, 0)$ such that $v > max(pathset(1, v))$. The first cycle is initialized with $p(1)$ and each time a maximal node is encountered a new cycle starts. Non maximal nodes are inserted in the current cycle.

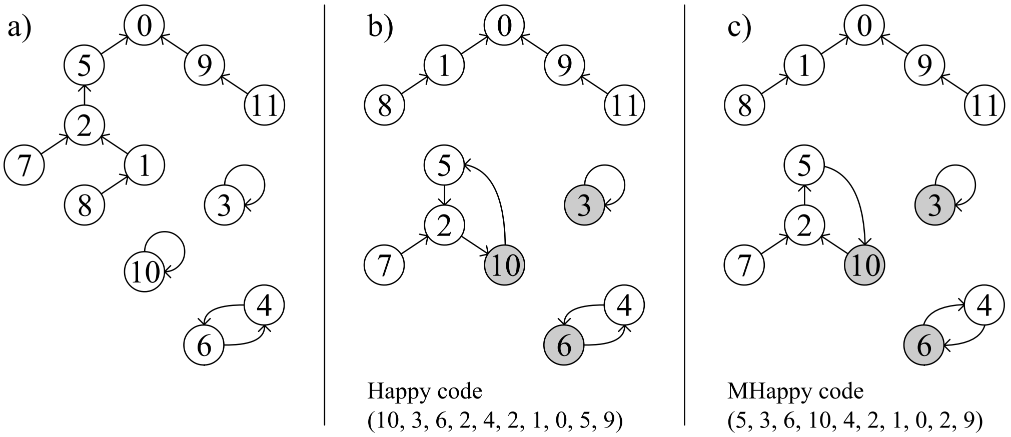
The encoding algorithm has been described in [89] as follows:

HAPPY ENCODING ALGORITHM

```
Root $T$ in $0$
Initialize $J = p(1)$
while $p(1) \neq 0$ do                      // consider $pathset(1,0)$ until $p(1) = 0$
  $j = p(1)$
  delete $(1, j)$                           // detach $j$
  delete $(j, p(j))$ and add $(1, p(j))$    // attach 1 above $j$
  if $j > J$ then                           // if $j$ is maximal
    $J = j$                                 // start a new cycle
    add $(J, J)$
  else                                      // insert $j$ in the current cycle close to $J$
    add $(j, p(J))$
    delete $(J, p(J))$
    add $(J, j)$
  for $v = 2$ to $n$ do
    $C[v-2] = p(v)$                         // compute the codeword
```

Figures 5.3a and 5.3b show an example of the HAPPY ENCODING ALGORITHM applied to the tree T of Figure 5.2a. Notice that the algorithm inserts a node j in a cycle immediately after J, the maximal node in the cycle. This implies that nodes in a cycle of the resulting graph are in reversed order with respect to their position in the original tree, e.g., in Figure 5.3b the edge $(5, 2)$ was $(2, 5)$ in the original tree. With the intent of keep the digraph as close as possible to the original tree, we now introduce a slightly modified version of this code which avoids this inversion: j is attached immediately before J instead of immediately after it. Let us call this modified version of Happy code *MHappy* code. The resulting digraph obtained applying the MHappy code to T is shown in Figure 5.3c.

The HAPPY ENCODING ALGORITHM works in $O(n)$ running time since it requires a number of steps equal to the length of the path between 1 and 0 in the tree.



Happy code
(10, 3, 6, 2, 4, 2, 1, 0, 5, 9)

MHappy code
(5, 3, 6, 10, 4, 2, 1, 0, 2, 9)

Figure 5.3: a) An intermediate step of the execution of the HAPPY ENCODING ALGORITHM on T of Figure 5.2a; b) the end of the execution and the resulting codeword, maximal nodes are represented in grey. c) Graph and codeword obtained with the MHappy code.

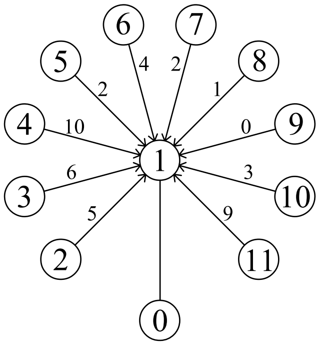
5.2.3 Dandelion Code

The Dandelion code is equivalent to the E-R Bijection introduced in [43] by Eğecioğlu and Remmel, but is described by Picciotto in a totally different way. It encodes the tree recursively attaching all nodes to node 1. During this process labels on edges are introduced, these labels will be used to create the code.

The encoding algorithm for Dandelion code, as presented in [89], is the following:

DANDELION ENCODING ALGORITHM

```
Root $T$ in $0$
for $v = n$ down to $2$ do
  $h = p(v)$
  $k = p(1)$
  delete $(v, h)$
  add $(v, 1)$ with label $label(v, 1) = h$
  if a cycle has been created then
    delete $(1, k)$
    add $(1, h)$
```



Dandelion code
(5, 6, 10, 2, 4, 2, 1, 0, 3, 9)

Figure 5.4: The tree T of Figure 5.2a after the execution of the DANDELION ENCODING ALGORITHM.

```
10.         $label(v, 1) = k$
11. for $v = 2$ to $n$ do
12.     $C[v - 2] = label(v, 1)$
```

The poetic name Dandelion derives from the fact that connecting all the nodes to node 1, a graph which looks like a dandelion flower is created (see Figure 5.4.

Testing if a cycle has been created (line 7) is the most expensive operation required at each step. A straightforward implementation of this algorithms requires $O(n^2)$ running time.

Improvements

Analogously to what has been done for the Blob code, we here give a characterization of all those node that satisfy the test in line 7, let us call them *flying* nodes. This characterization allow us to precompute, for each node, whether it satisfies the test or not, yielding a linear time algorithm.

Lemma 5.7. *Flying nodes are all nodes v such that $v \in pathset(1, 0)$ and $v > max(pathset(v, 0))$.*

Proof. The first condition trivially holds, otherwise cycles cannot be created.

Given $v \in pathset(1,0)$, let $m = max(pathset(v,0))$. If $m > v$ then m is processed before v by the algorithm, m is directly connected to 1 and it introduces a cycle containing v. When the cycle is broken (line 8 and 9), all the nodes in the cycle are excluded by the resulting $pathset(1,0)$. This implies that in successive steps v cannot be a flying node.

On the other hand, if $v > m$ it will be in $pathset(1,0)$ when it is processed by the algorithm and so it obviously introduces a cycle. □

The condition stated in Lemma 5.7 allows us to easily precompute whether a node satisfies the test in line 7 with a simple scan of the path from 1 to 0. This decreases the running time of the DANDELION ENCODING ALGORITHM to $O(n)$.

5.3 E-R Bijection

In [43] Eğecioğlu and Remmel introduce a bijection, called θ_n, that associates functions in $[1, n-2] \rightarrow [0, n-1]$ with Cayley trees (labeled in $[0, n-1]$ rooted in the fixed node $n-1$)[1]. This bijection can be straightforward interpreted as a code for labeled trees. Indeed a function in $[1, n-2] \rightarrow [0, n-1]$ can be written as a sequence of $n-2$ numbers in $[0, n-1]$. When it is used as a code, θ_n is often called E-R Bijection.

Give a function $f : [1, n-2] \rightarrow [0, n-1]$, this bijection uses it to build a graph $G([0, n-1], (i, j) : f(i) = j)$, i.e. the functional digraph associated to $g : [0, n-1] \rightarrow [0, n-1]$ defined as follows:

$$g(i) = \begin{cases} undefined & \text{if } i = 0 \\ undefined & \text{if } i = n-1 \\ f(i) & \text{otherwise} \end{cases}$$

G will necessarily contains two trees rooted at node 0 and $n-1$, respectively. All other connected components in G have a cycle as core (see Corollary 5.3).

According with the original definition of θ_n, the graph is drawn so that:

[1] In the original presentation authors used labels in $[1, n]$ and n as root.

a)

i:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$f(i)$:	4	3	4	2	20	6	11	0	3	3	19	18	18	5	0	15	5	6	11

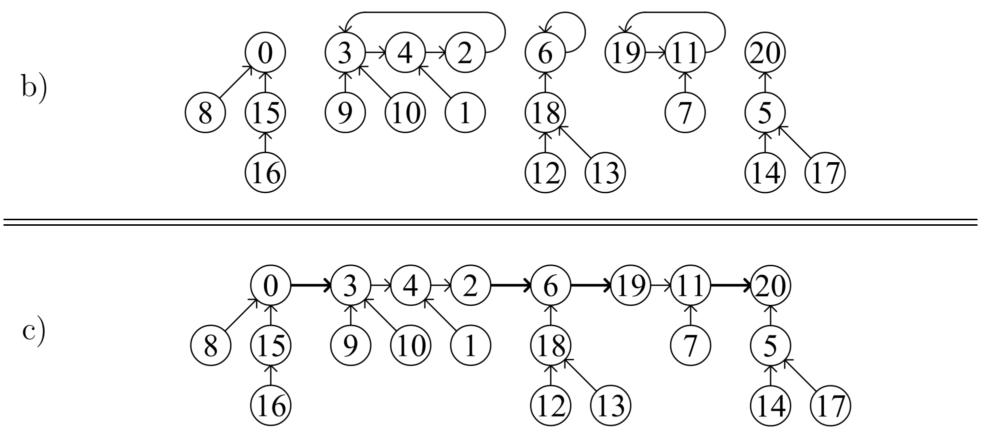


Figure 5.5: a) A function $f : [1, 19] \to [0, 20]$; b) the digraph G corresponding to f drawn according to rules 1-3; c) the tree $T = \theta_{21}(G)$.

1. the tree rooted at 0 and $n-1$ are drawn on the extreme left and extreme right respectively;
2. cycles are drawn so that their edges form a directed path on the line between 0 and $n-1$ with one backedge above the line;
3. each cycle is arranged so that its smallest node is on the right and the cycles are ordered from left to right by increasing smallest node.

In Figure 5.5a and Figure 5.5b a function and the corresponding digraph are shown. The digraph is draw according to rules 1-3.

Now, in order to obtain a tree we must break all cycles and join all connected components. Let us call r_i and l_i the rightmost and the leftmost node in each cycle according to the described drawing. θ_n deletes all backedge (r_i, l_i) and inserts the edges $(0, l_1), (r_1, l_2), (r_2, l_3), \ldots, (r_{k-1}, l_k), (r_k, n-1)$, where k is the number of cycles in G. In Figure 5.5c the obtained tree is shown.

The inverse θ_n, (i.e., the tree encoding), can be obtained directly from its definition: consider the path between 0 and $n-1$, each node $v <$

$min(pathset(v, n-1))$ will be a right node of a cycle. This is all we need to correctly split the path between 0 and $n-1$ in cycles. Eğecioğlu and Remmel also generalized their bijection so that any node can play the role of 0 and n-1.

The idea of considering the codeword as the list of a function associated to a functional digraph has inspired our general scheme described below.

5.4 Functional Digraph Transformation

The easiest method to associate a rooted Cayley tree with a string is to list, for each node, its parent: this is the naïve code (see Section 4.2). If the tree is always rooted in a fixed node (0 in our case) the resulting codeword has length $n-1$ and then this method is not bijective.

In this section we present a general scheme for defining bijections between the set of labeled trees with n nodes and the set of codewords of length $n-2$. Our idea is to modify the naïve code to reduce the length of the codeword that it yields.

If the tree is rooted in a fixed node x, and there exists another fixed node y having x as parent, the length of the naïve code can be reduced to $n-2$ omitting the information related to both x and y. It is easy to root a given unrooted tree at a fixed node x, while it is not clear how to guarantee the existence of edge (x, y). For this reason our general scheme is parametrized by a function φ that manipulates the tree in order to ensure the existence of (x, y). Parameters φ, x, and y characterize each specific instance of our general scheme.

In order to be suitable for our general scheme the function φ must satisfy certain constraints. It must transform the tree into a graph such that:

1. x has no outgoing edges;
2. the only outgoing edge of node y is (x, y) ;
3. each other node has exactly one outgoing edge.

these constraints guarantee that a codeword of length $n-2$ can be generated listing the endpoint of the outgoing edge of each node but x and y.

The three constraints listed above imply that φ is a function that transforms a tree T into a functional digraph G. G has $n-1$ edges and corresponds to a function g such that $g(x)$ is undefined and $g(y) = x$. We will call the class of such graphs $\mathcal{F}_n^{xy}$.

Lemma 5.8. $|\mathcal{F}_n^{xy}| = n^{n-2} = |\mathcal{T}_n|$

Proof. Consider all functions $h : [0, n-1] \smallsetminus \{x, y\} \rightarrow [0, n-1]$, there are clearly n^{n-2} such functions. For each h we can derive a digraph in $|\mathcal{F}_n^{xy}|$ associated to a function g defined as follows:

$$g(i) = \begin{cases} undefined & \text{if } i = x \\ x & \text{if } i = y \\ h(i) & \text{otherwise} \end{cases}$$

Thus $|\mathcal{F}_n^{xy}| \geq n^{n-2}$.

On the other hand, for each digraph in $|\mathcal{F}_n^{xy}|$ associated to a function g, we can univocally identify a function h such that $h(i) = g(i)\ \forall i \in [0, n-1] \smallsetminus \{x, y\}$. This implies $|\mathcal{F}_n^{xy}| \leq n^{n-2}$.

$\square$

We now present the general encoding scheme when φ, x, and y are given:

GENERAL ENCODING SCHEME

```
Root $T$ in $x$
Construct $G = \varphi(T)$
for $v = 0$ to $n - 1$ do
    if $v \neq x$ and $v \neq y$ then
        append $g(v)$ to $C$
```

In order to guarantee the bijectivity of the obtained encoding, the function φ must be invertible, i.e. there must exist a function φ^{-1} that, given a functional digraph G in $\mathcal{F}_n^{xy}$, is able to revert it to the tree T corresponding to G, i.e., $\varphi(T) = G$.

If φ^{-1} exists, it is possible to define the general decoding scheme:

GENERAL DECODING SCHEME

```
for $v = 0$ to $n - 1$ do
   if $v \neq x$ and $v \neq y$ then
      extract the first element $u$ from $C$
      $g(v) = u$
$g(x) = undef$
$g(y) = x$
Reconstruct the graph $G$ from $g$
Compute $T = \varphi^{-1}(G)$
```

In the following we will show how to map Blob code, MHappy code and Dandelion code into our scheme. For each code we will define a φ function to compute it, we will provide an inverse function to decode it, and we will discuss the running time required for both encoding and decoding. Linear time algorithms for Happy and Dandelion code have been presented by Picciotto in terms of string permutations. We rather focus on graph algorithms because it helps us to understand why these codes have good locality and heritability.

5.4.1 Blob Transformation

In this section we introduce a function φ_b suitable to map the Blob code presented in Section 5.2.1 into our GENERAL ENCODIGN SCHEME. We will exploit the characterization of stable nodes given above. The function φ_b constructs a graph G from a tree T in the following way: for each unstable node v, remove edge $(v, p(v))$ and add edge $(v, p(z))$ where $z = min\{u : u > v \text{ and } u \text{ unstable}\}$. If z does not exist (i.e., if $v = n - 1$), add edge $(v, 0)$.

In Figure 5.6a all stable nodes (i.e., all v such that $v < max(pathset(v, 0))$) of the tree T depicted in Figure 5.2a are marked in gray. Figure 5.6b shows

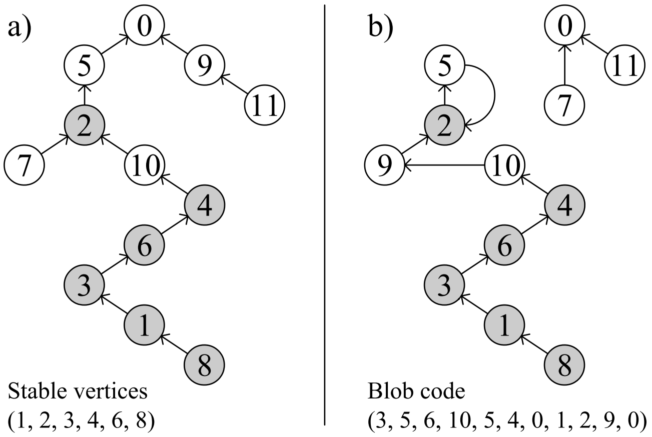


Figure 5.6: a) The tree T of Figure 5.2a with stable nodes identified in grey; b) $G = \varphi_b(T)$ and the Blob code representing T.

the digraph obtained transforming T by means of φ_b.

Before proving that φ_b allows us to correctly compute the Blob code we need to prove the following lemma:

Lemma 5.9. *Each path in T from a stable node v to $m = max(pathset(v, 0))$ is preserved in $G = \varphi_b(T)$.*

Proof. Let v be a stable node. Let us assume by contradiction that the path from v to $m = max(pathset(v, 0))$ is in T but not in $G = \varphi_b(T)$. This means that in the transformation from T to G at least one node w in $pathset(v, m)$ has changed its parent. Since φ_b changes only edges outgoing from unstable nodes, w should be unstable and then $w > max(pathset(w, 0))$. $w \in pathset(v, m)$ implies $m \in pathset(w, 0)$, then w should be greater than m contradicting $m = max(pathset(v, 0))$. □

Theorem 5.10. *Blob code fits into the General Scheme when $x = 0$, $y = n - 1$, and $\varphi = \varphi_b$.*

Proof. It is easy to see that graph $G = \varphi_b(T)$ is a functional digraph, since each node has outdegree equal to 1. Moreover the function g associated with G is undefined in 0 and $g(y) = 0$, by definition of φ_b.

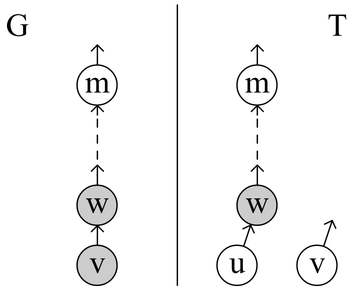


Figure 5.7: Nodes involved in the proof of Theorem 5.10 both in G and in T. Stable nodes are represented in grey.

Lemmas 5.5 and 5.6 guarantee that the generated codeword C is equal to the codeword computed by BLOB ENCODING ALGORITHM.

Now we have to prove that φ_b is invertible, i.e. we have to show how to rebuild T from G: all cycles in G must be broken, and stable and unstable nodes recomputed.

Each cycle Γ is broken deleting the edge outgoing from γ, the maximum label node in Γ. Lemma 5.9 implies that γ unstable in T, otherwise a path from γ to $max(pathset(\gamma, 0))$ would have been preserved in G, then a node greater than γ would appear in Γ. Notice that γ becomes the root of its own connected component, while 0 is the root of the only connected component not containing cycles. The identification of γ is a step towards the recomputation of stable and unstable nodes.

We call stable in G each node v such that $v < max(pathset(v, \gamma_v) \bigcup \{\gamma_v\})$, where γ_v is the root of the connected component containing v. Lemma 5.9 guarantees that each node v stable in T, is also stable in G. Now we prove that vice versa is also true.

Let us assume, by contradiction, that there exists a node v stable in G but unstable in T. Let $m = max(pathset(v, \gamma_v) \bigcup \{\gamma_v\})$ in G. It holds $v < m$ and m unstable both in G and in T. In G node m is unstable because there is no node greater than m in $pathset(v, \gamma_v) \bigcup \{\gamma_v\}$; in T node m cannot be stable because, as noted before, each stable node in T remains stable in G.

W.l.o.g. let us assume that all nodes between v and m are stable both in G and in T. Let w be the parent of v in G. By definition of φ_b there exists a node $u > v$ unstable in T such that $p(u) = w$ in T. In Figure 5.7 v, m, u, and w are depicted both in G and in T.

Since m is in the path from u to 0 in T and u is unstable in T, m must be smaller than u. Then $v < m < u$ and m is unstable in T, this is impossible because φ_b chooses u as the smaller unstable node greater than v in T.

Once stable and unstable node are correctly identified, it is straightforward to rebuild the tree $T = \varphi_b^{-1}(G)$. □

Linear running time algorithms for both encoding and decoding can be obtained by fitting Blob code into our General Scheme. Indeed both φ_b and φ_b^{-1} can be implemented in $O(n)$ sequential time: computation of the maximum node in the upper path (coding) and cycles identification (decoding) can both be implemented by simple search techniques.

Let us now show a possible implementation:

```
function ComputeUpperMax($v$)
if $upperMax(p(v))$ is undefined then
    ComputeUpperMax($p(v)$)
$upperMax(v) = max\{p(v), upperMax(p(v))\}$
```

```
COMPUTE $\varphi_b$
$upperMax(0) = 0$
for $v = 1$ to $n - 1$ do
    if $upperMax(v)$ is undefined then
        ComputeUpperMax($v$)
$prevNonStatic = n - 1$
for $v = n - 2$ down to $1$ do
    if $upperMax(v) > v$ then                    // if $v$ is a static node
        $g(v) = p(v)$
    else                                          // a non static node
        $g(v) = p(prevNonStatic)$
        $prevNonStatic = v$
```

```
$g(n-1) = 0$
$g(0) = undefined$
return $G$ corresponding to $g$
```

This algorithm begins with the computation of $upperMax(v)$, i.e. for each node v it computes $max(pathset(v, 0))$. Then this information is exploited to distinguish static and non static nodes and to assign the correct value to $g(v)$. Using this algorithm to compute φ_b together with the GENERAL ENCODING SCHEME introduced in Section 5.4, it is possible to compute the Blob code for a tree in $O(n)$ time.

The decoding process is a bit more difficult, but still requires linear time. The hardest part in the computation of φ_b^{-1} is the computation of $max(pathset(v, \gamma_v) \bigcup \{\gamma_v\})$ for each node v, let us call this information $\mu(v)$. This can be done without explicitly identifying γ_v and then without breaking cycles. In order to avoid the risk of infinite recursion on cycles we will associate to each node a variable *status* to distinguish whenever a node is still being processed or not.

function ComputeMu(v)

```
 1. if $status(p(v)) = processed$ then                  // no need to iterate
 2.    $\mu(v) = max\{\mu(p(v)), p(v)\}$
 3.    $status(v) = processed$
 4. else                                                 // iterate
 5.    $status(v) = inProgress$
 6.    if $status(p(v)) = inProgress$ then               // this is a cycle
 7.       $\mu(v) =$ MaxInCycle($v$)
 8.    else
 9.       ComputeMu($p(v)$)                              // recursive call
10.       $\mu(v) = max\{\mu(p(v)), p(v)\}$
11.    $status(v) = processed$
```

Let us clarify the condition of line 6 in function ComputeMu. The status of a node v is *inProgress* only during a recursive call of ComputeMu on the ascending path of v. If the condition of line 6 is true, a cycle has been identified, i.e. following outgoing edges we move from v back to v itself. At this point, in order to avoid infinite recursion on cycles, an auxiliary function to compute the maximum in this cycle is called and the recursion terminates. The auxiliary function MaxInCycle simply follows outgoing edges from v until it comes back on v and returns the maximum label encountered.

COMPUTE φ_b^{-1}

```
$\mu(0) = 0$
$status(0) = processed$
for $v = 1$ to $n-1$ do
   if $status(v) \neq processed$ then
      ComputeMu($v$)
$prevNonStatic = n-1$
for $v = n-2$ down to $1$ do
   if $\mu(v) > v$ then                      // if $v$ is a static node
      $p(v) = g(v)$
   else                                       // a non static node
      $p(prevNonStatic) = g(v)$
      $prevNonStatic = v$
$p(prevNonStatic) = 0$
$p(0) = undefined$
return $T$ corresponding to $p$
```

Since function $\mu(v)$ can be computed, for each v, in overall linear time with function ComputeMu, φ_b^{-1} requires $O(n)$ running time.

The experimental analysis presented in [61] shows that locality and heritability are satisfied by the Blob code much better than by the Prüfer code. The reasons behind this experimental result become clear when Blob code is analyzed according to our method, which is quite different from Picciotto's description. The functional digraph generated by φ_b preserves an edge of the original tree for each stable node, and for these nodes $g(v) = p(v)$: this partial similarity with naïve code is the reason why Blob code exhibits good locality and heritability. We recall that naïve code has maximal locality and heritability (see Section 4.2).

In the next two sections we will see how MHappy code and Dandelion code preserve similarities with naïve code more that Blob code.

5.4.2 MHappy Transformation

Here we show how to map our Modified Happy code introduced in Section 5.2.2 into our general scheme. The same result for the original Happy code can be obtained analogously.

We define a function φ_m which, given a tree T, constructs a graph G by considering only the path from 1 to 0. For each maximal node v in $pathset(1,0)$ remove the edge incoming in v, and add an edge (z,v) where z is a node in $pathset(v,0)$ such that $p(z)$ is the smaller maximal node greater than v. If z does not exist, use the child of 0; finally remove the edge incoming in 0 and add the edge $(1,0)$.

Theorem 5.11. *MHappy code fits into the General Scheme when $x = 0$, $y = 1$, and $\varphi = \varphi_m$.*

Proof. It is trivial to see that MHappy encoding transforms T into the same functional digraph generated by φ_m: this corresponds to a function g undefined in 0 and is such that $g(1) = 0$.

To show that φ_m is invertible, first sort all cycles in G in increasing order with respect to their maximum node γ, then break each cycle removing the edge incoming in γ. Since the order of cycles obtained is the same as that in which they were originally created, we rebuild the original tree inserting all the nodes of each cycle in the path from 1 to 0 according to the order of the cycles. □

We now detail the implementation of φ_m and φ_m^{-1}.

COMPUTE φ_m

```
1. $g(v) = p(v) \quad \forall v \in [0, n-1]$
2. identify all maximal nodes $m_1, m_2, \ldots, m_k$ append $m_{k+1} = 0$
3. identify their predecessors $pred(m_1), \ldots, pred(m_{k+1})$
4. for $i = 1$ to $k$ do
5.     $last(m_i) = pred(m_{i+1})$          // the last element of $m_i$'s cycle
6.     $g(last(m_i)) = m_i$                 // close each cycle
7. $g(1) = 0$
8. return $G$ corresponding to $g$
```

Computations of lines 2 and 3 can be easily achieved with a simple scan of the path from 1 to 0, then the algorithm requires linear time.

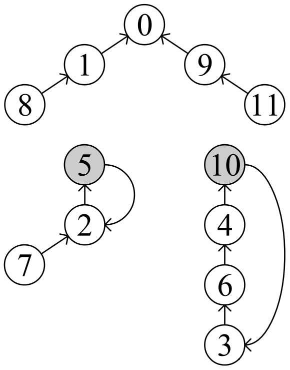
To compute φ_m^{-1} we exploit the same $\mu(v)$ computed in Section 5.4.1, indeed it is easy to see that $\mu(v) = v$ if and only if v is the maximum node in a cycle.

COMPUTE φ_m^{-1}

```
1. $p(v) = g(v) \quad \forall v \in [1, n-1]$
2. identify all nodes $m_1, m_2, \ldots, m_k$ such that $\mu(m_i) = m_i$
3. identify their predecessors $pred(m_1), \ldots, pred(m_k)$ in cycles
4. $p(1) = m_1$
5. for $i = 1$ to $k-1$ do
6.     $p(pred(m_i)) = m_{i+1}$             // break each cycle
7. $p(pred(m_k)) = 0$
8. return $T$ corresponding to $p$
```

$\mu(v)$ can be computed, for each v, in linear time, as shown in Section 5.4.1 and line 3 requires nothing more than a scan of each cycle. Therefore encoding and decoding MHappy code requires $O(n)$ running time; the same bound holds for the original Happy code.

We underline that φ_m modifies only a subset of the edges on the path between 1 and 0, so it preserves the topology of T much better than φ_b. For



Dandelion code
(5, 6, 10, 2, 4, 2, 1, 0, 3, 9)

Figure 5.8: The digraph $G_D = \varphi_d(T)$, flying nodes are represented in grey.

this reason in [25] we claimed that MHappy code should satisfy locality and heritability properties better than Blob code.

5.4.3 Dandelion Transformation

In this section we map the Dandelion code into our general scheme. Let us consider the DANDELION ENCODING ALGORITHM and the characterization of flying nodes given in Section 5.2.3.

We define a function φ_d that transforms T in G considering only flying nodes in decreasing order. For each flying node v, φ_d swaps $p(v)$ and $p(1)$ (see Figure 5.8).

Theorem 5.12. *Dandelion code fits into the General Scheme when $x = 0$, $y = 1$, and $\varphi = \varphi_d$.*

Proof. $G = \varphi_d(T)$ is a functional digraph corresponding to a function g undefined in 0 and such that $g(1) = 0$. The code generated using φ_d is the same as that using DANDELION ENCODING ALGORITHM. Indeed when the algorithm breaks a cycle in a flying node v, node 1 is connected to h (the former parent of v) and the label of edge $(v, 1)$ becomes k (the former parent of 1). In code C, the position corresponding to v contains the value k. Non-flying nodes simply use their parents as edge labels. Thus, assigning $p(v) = k$

for flying nodes, it is possible to avoid edge labels. In this way, each non-flying node simply retrains its parent while each flying node exchanges its parent h with the parent of node 1, i.e., k. Since the DANDELION ENCODING ALGORITHM considers nodes in decreasing order, φ_d produces the same code.

To invert φ_d we again have to break cycles in the functional digraph corresponding with a given code. Flying nodes are all and only maximal nodes in cycles. Note that, in order to correctly rebuild the path from 1 to 0, cycles of G must be considered in increasing order of their maximum node. □

The algorithms obtained fitting Dandelion code into our general scheme are linear. Indeed the computation of φ_d and φ_d^{-1} requires the same operations used for φ_m and φ_m^{-1}. Let us detail the implementation of both φ_d and φ_d^{-1}:

COMPUTE φ_d

```
1. $g(v) = p(v) \quad \forall v \in [0, n-1]$
2. identify all flying nodes $f_1, f_2, \ldots, f_k$ in path from 1 to 0
3. for $i = 1$ to $k$ do
4.   swap $g(1)$ and $g(f_i)$
5. return $G$ corresponding to $g$
```

Since line 2 requires a simple backward scan of the path from 1 to 0, the algorithm requires $O(n)$ time.

The implementation of φ_m^{-1} relies on the fact that flying nodes are all and only maximal nodes in cycles. We already know that for these nodes it holds $\mu(v) = v$.

COMPUTE φ_d^{-1}

```
1. $p(v) = g(v) \quad \forall v \in [1, n-1]$
2. identify all nodes $f_i$ such that $\mu(f_i) = f_i$
3. sort $f_1, f_2, \ldots, f_k$ in decreasing order
```

```
for $i = 1$ to $k$ do
    swap $g(1)$ and $g(f_i)$
return $T$ corresponding to $p$
```

In view of the considerations made in the previous sections regarding the computation of μ, this algorithm runs in linear time.

5.5 Comparing Transformation Codes

Both MHappy code and Dandelion code modify only a subset of the edges on the path between 1 and 0. Even the E-R Bijection focuses on a single path of the tree and splits it into cycles. Moreover, if in the E-R Bijection we use 1 instead of 0 and 0 instead of $n-1$, the resulting encoding is almost identical to the Dandelion code. The unique difference is that in the Dandelion code flying nodes are such that $v > max(pathset(v, 0))$, while in the E-R Bijection rightmost nodes satisfy the inequality $v < min(pathset(v, 0))$. Happy code and MHappy code, instead, focus on the $pathset(1, v)$ so they basically analyze the path in the reverse direction. Indeed, as Picciotto has proved, given a tree T, if we obtain T' reverting the order of nodes in the path between 1 and 0, the Happy code for T coincides with the Dandelion code for T' and vice versa.

In a paper we published in 2005, we conjectured that MHappy code and Dandelion code should have similar locality and heritability, and that both of them are better than Blob code. Later then, in 2006, experimental comparisons of these codes in GA have been made by Paulden and Smith [85]. They also tested all possible variations of codes that split a fixed path considering foreward/backward minimum/maximum nodes in increasing/decreasing order. They called them Dandelion-like codes. Their experiments show that, with respect to performances in GA, these codes are splitted in two groups. Paulden and Smith concluded that: *"The Group 2 codings (including the MHappy Code) were found to have similar locality properties and slightly better heritability properties to the Group 1 codings (including the Dandelion Code and Happy Code)"* [85].

Paulden and Smith also proved that Dandelion code has asymptotically optimal locality [84].

5.6 Concluding Remarks

The General Scheme introduced in this section is suitable to interpreter known codes as transformations between trees and functional digraphs. It gives us a better comprehension of how encoding preserves the topology of the tree, and therefore it helps to understand which code better fits desirable properties, such as locality and heritability.

As we have seen, this General Scheme is suitable to code unrooted trees considering them as rooted in a fixed node y. In order to code arbitrarily rooted trees, it is enough to add the root label to an $n-2$ length codeword (e.g., as a last symbol) obtained re-rooting the tree in y. In this way the scheme encodes a rooted tree with n nodes with codewords of length $n-1$, that is still a bijective mapping. If both x and y are added to the codeword, each instance of the General Scheme becomes an implementation of the Joyal [60] bijection between vertebrates (doubly-rooted labeled trees) and strings of length n.

The class of Transformation codes (i.e., those codes that can be mapped into our general scheme) contains each possible bijective code for labeled trees. This is not true for other classes such as Prüfer-like codes (which are based on recursive leaves elimination). As an example consider the tree T of Figure 5.2a. The Dandelion code for T is $C_D = (5, 6, 10, 2, 4, 2, 1, 0, 3, 9)$, it is not possible to find a Prüfer-like code able to associate T with C_D, indeed at the first step, among all leaves, no one has 5 as parent.

On the other hand, it is possible to define a function φ_p that yields the Prüfer code through the GENERAL ENCODING SCHEME for any chosen x and y. Let C_P be the Prüfer code associated to T, then functional digraph $G_P = \varphi_p(T)$ should have an edge (x, y) and no edges outgoing from y. All other edges can be derived directly from C_P: consider the set $V - x - y$ in increasing order, the i-th node v_i gets the outgoing edge $(v_i, C_P[i])$. Figure 5.9

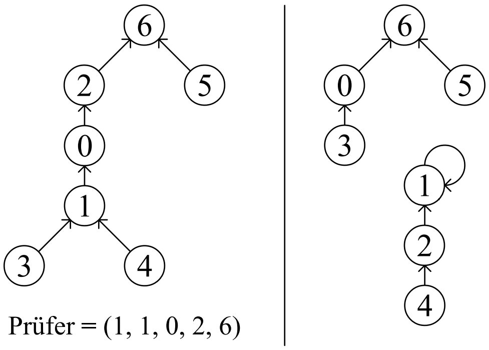


Figure 5.9: A sample tree T and its Prüfer code together with the digraph $G_P = \varphi_p(T)$; we used $x = 0$ and $y = 6$.

shows an example of applying φ_p, we have chosen $x = 0$ and $y = n - 1$. In this example almost all edges have been changed, then the topology of the functional digraph is completely different from the one of the tree, this confirms the poor locality of Prüfer code.

This idea can be easily exploited to map any possible bijective code to our General Scheme. In the second part of this thesis we will see how bijective tree codes can be extended to the class of k-tree.

Part II

Generalizations

Chapter 6

Encoding k-Trees

In this chapter we consider bijective codes for the class of k-trees, i.e., one of the most natural and interesting generalizations of trees (for a formal definition see Section 6.1). 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 [8, 9, 10]. Moreover each k-tree is also a minimal k-connected graph and thus a minimal k-fault tolerant network [56].

In 1970 Rényi and Rényi [94] 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 codewords was produced [28, 46]. Attempts have been made to obtain an algorithm with linear running time for the redundant Prüfer code [73].

As an original contribution of this thesis we present a novel bijective code for k-tree. We also give a detailed description of linear time encoding and decoding algorithms for our code. It is worth mention that our code can be easily adapted to rooted, unrooted, and Rényi k-trees, always preserving bijectivity. To the best of our knowledge, no linear time algorithms for encoding and decoding k-tree have been presented in the literature before our work [22, 23].

The chapter is organized as follows: in Section 6.1 we provide definitions and combinatorial results on k-trees. In Section 6.2, we survey known bijective codes for k-tree. In Section 6.3 and 6.4 we introduce two building blocks of our coding technique (Characteristic Tree and Generalized Dandelion Code) while all details for the encoding and decoding algorithm are given in Section 6.5. In Section 6.6 we discuss the physical representation of our code. The chapter ends with some conclusions and future directions for research in this topic.

6.1 Preliminaries

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

Definition 6.1. [57] 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 $T_k = (V \cup \{v\}, E \cup \{(v, x) \,|\, x \in K\})$ 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 6.2. 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

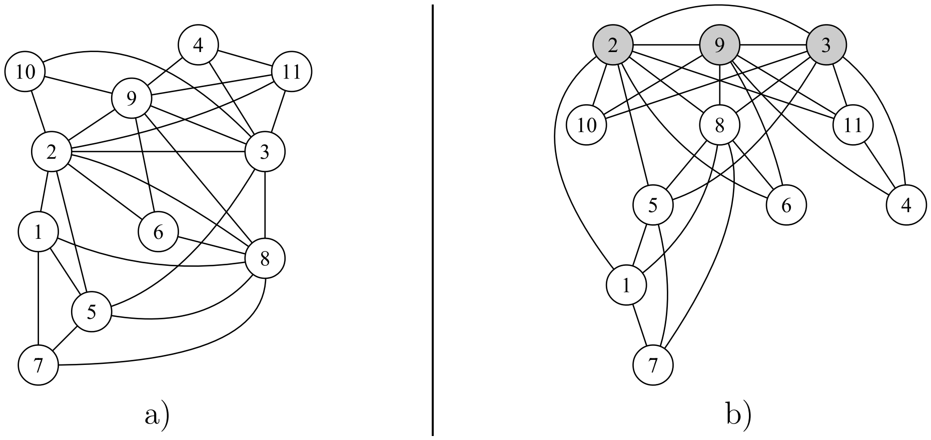


Figure 6.1: a) An unrooted 3-tree T_3 with 11 nodes; b) T_3 rooted in the clique $\{2, 3, 9\}$.

to R (see [94]). Since k-trees are perfect elimination order graphs [95], 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 single clique, at most one node adjacent to the removed k-leaf may become a k-leaf.

In Figure 6.1(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 6.1(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 [6, 47, 77, 94]):

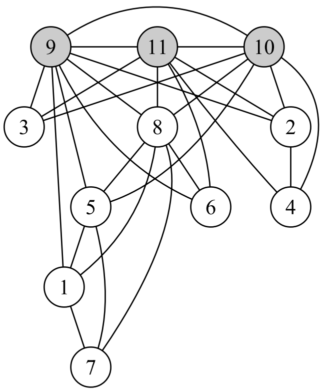
$$|\mathcal{T}_k^n| = \binom{n}{k} (k(n-k)+1)^{n-k-2}$$

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 (see Chapter 2).

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:

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

Definition 6.3. [94] A Rényi k-tree R_k is a rooted k-tree with n nodes labeled in $[1, n]$ and root $R = \{n-k+1, n-k+2, \ldots, n\}$.



$$(\{9, 10, 11\}, \{2, 10, 11\}, \{9, 10, 11\}, \{1, 5, 8\}, \{5, 8, 9\}, \{8, 9, 10\})$$

Figure 6.2: A Rényi 3-tree R_3 with 11 nodes and root $\{9, 10, 11\}$ together with its redundant Prüfer code.

It has been proven [77, 94] 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 6.4. 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 considered a Rényi k-trees. This implies $\mathcal{R}_k^n \subseteq \mathcal{T}_k^n$.

6.2 Known Codes

In 1970 Rényi and Rényi [94] generalized Prüfer's bijective proof of Cayley's theorem to code Rényi k-trees. Their code recursively eliminate from the k-tree the smallest k-leaf. Each time a k-leaf a is removed the set B of its adjacent nodes (that form a k-clique) is added to the codeword as a single symbol of the string. Nodes belonging to the root $R = \{n-k+1, n-k+2, \ldots, n\}$ are never considered as k-leaves. The procedure terminates when the codeword reaches length $n-k-1$.

Example 10. Consider the Rényi 3-tree in Figure 6.2. It has 3 k-leaves: $\{3, 4, 7\}$. At the beginning $a_1 = 3$ and $B_1 = \{9, 10, 11\}$. At the second step

$a_2 = 4$ and $B_2 = \{2, 10, 11\}$. When 4 is removed from the k-tree 2 becomes a k-leaf and therefore $a_3 = 2$ and $B_3 = \{9, 10, 11\}$. The algorithm proceeds removing 7, 1, 5, 6. The resulting codeword is $(\{9, 10, 11\}, \{2, 10, 11\}, \{9, 10, 11\}, \{1, 5, 8\}, \{5, 8, 9\}, \{8, 9, 10\})$ Notice that there is no need to remove 8 and add $B_{n-k} = R$ to the codeword, indeed, the last symbol is always the fixed root and then can be omitted (as it is for Cayley trees).

Notice that when this procedure is applied to Rényi 1-tree (i.e., simple Cayley trees rooted at n) it yields exactly the original Prüfer code.

The decoding is analogous to the one given by Prüfer: the leaf a_i removed at the i-th step of the encoding is deduced as the smallest number not yet used in $a_1, \ldots, a_{i-1}$ that does not appear in any subsequent symbol of the codeword, i.e., $B_i, \ldots, B_{n-k-1}$:

$$a_i = \min \left\{ a \in [1, n] \smallsetminus \{a_h\}_{h<i} \smallsetminus \bigcup_{j \geq i} B_j \right\}$$

This code is not bijective, because each symbol of the codeword is a set of k distinct elements in $[1, n]$ and then each codeword belongs to the set:

$$\binom{[1, n]}{k}^{n-k-1}$$

In order to obtain a correct counting result on the cardinality of the set of Rényi k-trees, in [94] the authors characterized all valid codewords. Unfortunately, computationally check whether a given codeword is valid or not is not immediate. From an algorithmic point of view, we can say that the validity conditions given by Rényi and Rényi basically correspond to an attempt of decode the codeword: if there is a step i where the algorithm is unable to deduce a_i the codeword is not valid, if the algorithm correctly reaches its end the codeword is valid.

From our perspective this is a severe shortcoming of this code: it cannot be used in those applications where a bijective code is required, like Random k-tree Generation and Genetic Algorithms over k-tree (topics of Chapter 4 naturally generalize to k-tree).

Subsequently, non redundant codes that realize bijection between k-trees (or Rényi k-trees) and a well defined set of codewords were produced by Eğecioğlu and Shen [46] and by Chen [28]. The latter one deals only with Rényi k-trees: the author motivates this choice saying that, as well as rooted trees are more natural than unrooted trees, Rényi k-trees are more natural than unrooted k-trees. We cannot agree with this claim, indeed, as discussed above, when $k > 1$ Rényi k-trees are a strict subset of unrooted k-trees. Thus we see this peculiarity of the Chen code as a limit. Moreover it does not seem that this code can be extended to obtain bijective codes for unrooted and arbitrarily rooted k-trees.

Eğecioğlu and Shen code

The work by Eğecioğlu and Shen is much more interesting than the one by Chen. They noticed that, since a k-tree has $K = k(n-k)+1$ cliques on k nodes and $K' = n-k$ cliques on $k+1$ nodes, the number of k-tree can be rewritten as:

$$|\mathcal{T}_k^n| = \binom{n}{k} K^{K'-2}$$

Their work relay on a generalization of the E-R Bijection: they interpreted $K^{K'-2}$ as a function f mapping all (but two) $(k-1)$-cliques to k-cliques. Each $(k-1)$-cliques is divided into $k+1$ "faces" (i.e., k-cliques), f describes how these faces should be composed to obtain the k-tree. The encoding relies on the existence of a complex orientation of all edges of the k-tree that induces an order among the faces of a $(k-1)$-clique and among the $(k-1)$-cliques themselves.

This code does not seem to admits efficient implementation. For these reason we devoted our efforts in designing a novel bijective code that admits linear time encoding and decoding.

6.3 Characteristic Tree

In this section we introduce the *characteristic tree* of a rooted k-tree. This is one of the building blocks of our code. We will use characteristic trees of Rényi k-trees in our coding process.

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

Definition 6.5. Given a rooted k-tree T_k with root R, obtainable by T'_k rooted at R by adding a new node v connected to a k-clique K (see Definition 6.1), 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 it holds for all $v \in K$, there always exists a unique node Y containing K at minimum distance from the root.

Definition 6.6. 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 $K' \smallsetminus K$ is guaranteed by Definition 6.5. 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. Therefore at least one element of $x \in K'$ does not appear in K. Moreover x is unique because $|K'| = |K|$ and $K = K' \smallsetminus \{x\} \cup \{v'\}$.

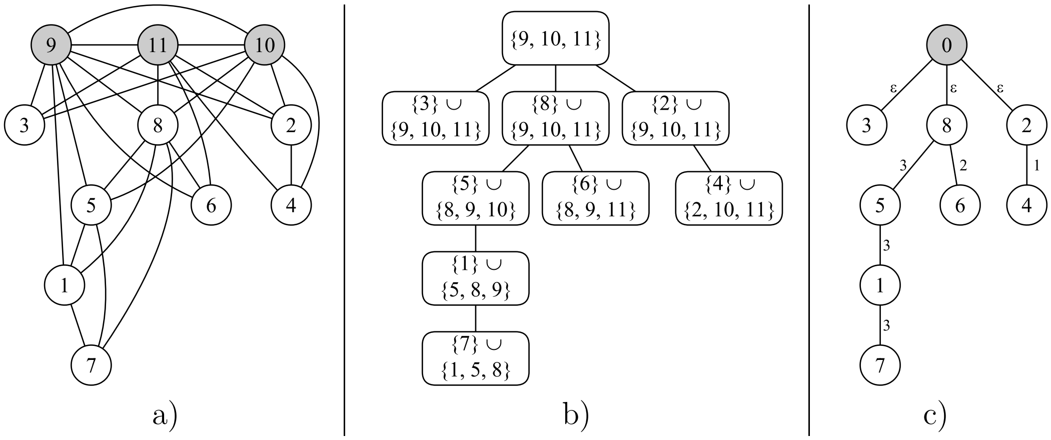


Figure 6.3: a) A Rényi 3-tree R_3 with 11 nodes and root $\{9, 10, 11\}$; b) the skeleton of R_3, with nodes $\{v\} \cup K$; c) the characteristic tree of R_3.

Remark 6.7. 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 6.3 a Rényi 3-tree with 11 nodes, its skeleton and its characteristic tree are shown.

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 tells 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 [52] and used in [28]. 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 have arbitrary labels in $[1, k]$. 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 6.4 we will show that $|\mathcal{Z}_k^n| = |\mathcal{R}_k^n|$.

6.4 Generalized Dandelion Code

In the first part of this thesis, many bijective codes for labeled trees have been presented. Here we show a generalization of the Dandelion code that takes into account labels on edges and can be used to encode characteristic trees of Rényi k-trees. We have arbitrarily chosen Dandelion code among several possible others[1]. We refer to Chapter 5 for a detailed description of Dandelion code. The approach we follow here is the one obtained through our GENERAL SCHEME (see Section 5.4.3).

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 encoding procedure T is rooted at r, thus identifying, for each node v, its parent $p(v)$. Considering T as a digraph with labeled oriented edges, the code recursively breaks the path between x and r into cycles until x reaches r. This is obtained by means of swap operations (see COMPUTE φ_d in Section 5.4.3).

We should specify what happen with edge labels when a swap takes place. Our algorithm ensures that the edge labels remain associated to parent nodes. More formally, when two nodes x and w swap their parents, 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 $(x, p(x))$.

[1] As discussed at the end of Section 6.5.1 also E-R Bijection, Happy code, and MHappy would have been valid choices.

The graph resulting from the encoding 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. The encoding algorithm is as follows:

GENERALIZED DANDELION ENCODING ALGORITHM

```
identify all flying nodes $f_1, f_2, \ldots, f_k$ in path from $x$ to $r$
for $i = 1$ to $k$ do
    $\ell(f_i, p(x)) = \ell(x, p(x))$
    $\ell(x, p(f_i)) = \ell(f_i, p(f_i))$
    swap $p(x)$ and $p(f_i)$
for $v \in V(T) \setminus \{r, x\}$ in increasing order do
    append $(p(v), \ell(v, p(v)))$ to the code
```

As for the Dandelion code, the running time of the encoding algorithm is $O(n)$.

In Figure 6.4 an example of Generalized Dandelion Encoding, with parameters $r = 0$ and $x = 1$, is presented.

As a further example let us encode (with $r = 0$ and $x = 1$) the tree shown in Figure 6.3(c). Here the only swap occurring is $p(1) \leftrightarrow p(8)$. The codeword obtained is: $[(0, \varepsilon), (0, \varepsilon), (2, 1), (8, 3), (8, 2), (1, 3), (5, 3)]$.

Remark 6.8. The Dandelion code satisfies Property 2.3, it is easy to extend this property to the Generalized Dandelion code. Consider the codeword

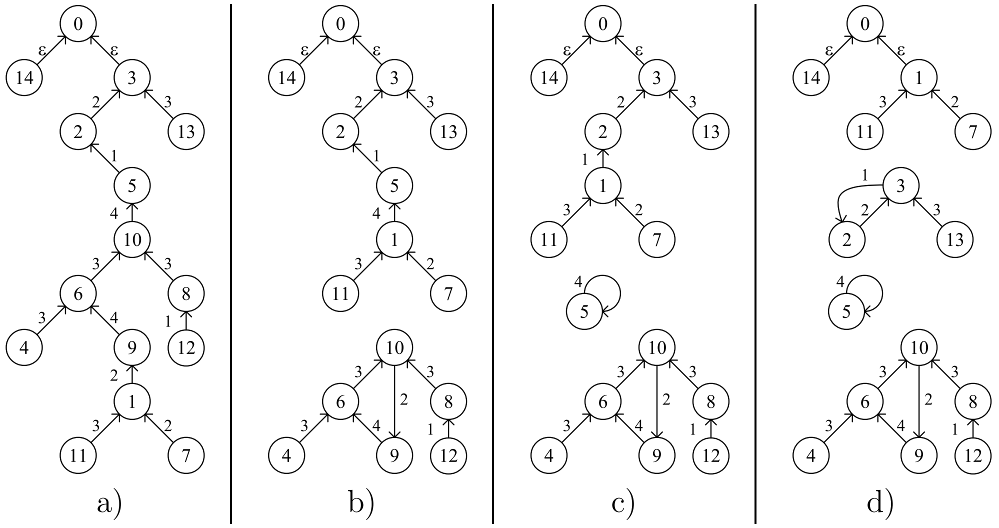


Figure 6.4: a) A tree T with 15 nodes labeled in $[0,14]$ and edge labels in $[1,4]$, represented as rooted at 0; b) after the first swap $p(1) \leftrightarrow p(10)$, cycle $\{10,9,6\}$ has been introduced; c) a loop 5 has been introduced, after the second swap $p(1) \leftrightarrow p(5)$; d) the tree T at the end of the encoding, after the last swap $p(1) \leftrightarrow p(3)$. The codeword is $[(3,2),(2,1),(6,3),(5,4),(10,3),(1,2),(10,3),(6,4),(9,2),(1,3),(8,1),(3,3),$ $(0,\varepsilon)]$.

associated to a tree T by the Generalized Dandelion code: all and only internal nodes of T appear (as first element of a pair) in the codeword.

Let us now detail how to decode a 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. At the end add the oriented edge (x,r) with label ε.

The decoding algorithms detailed below proceeds (as for the Dandelion code) breaking cycles in order to correctly reconstruct the path between x and r.

GENERALIZED DANDELION DECODING ALGORITHM

```
Construct $G$ from $S$
Identify all cycles in $G$ and their maximal nodes
for each maximal node $m_i$ in increasing order do
    $\ell(m_i, p(x)) = \ell(x, p(x))$
    $\ell(x, p(m_i)) = \ell(m_i, p(m_i))$
    swap $p(x)$ and $p(m_i)$
```

The algorithm retrains $O(n)$ running time. Indeed, managing edge labels only requires $O(1)$ extra operations at each step.

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 codeword 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 codewords is $(k(n-k)+1)^{n-k-1}$, and then $|\mathcal{Z}_k^n| = (k(n-k)+1)^{n-k-1} = |\mathcal{R}_k^n|$.

6.5 A New Code for k-Trees

In this section we present a new bijective code for k-trees and we detail, for this code, linear time encoding and decoding algorithms. To the best of our knowledge, this work is the first one that explicitly provides efficient algorithms to encode and decode k-trees.

6.5.1 Encoding Algorithm

Our algorithm initially transforms a k-tree into 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 encode R_k. The codeword for $T(R_k)$ is then modified (according with information related to Q) to obtain a codeword for T_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 performed with a running time linear in the size of the k-tree, i.e., $O(nk)$.

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 codewords 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 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.

Our bijective code for k-trees associated elements in $\mathcal{T}_k^n$ with elements in:

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

Note that $|\mathcal{A}_k^n| = |\mathcal{T}_k^n|$. In the next section we will describe a decoding process that is able to associate each codeword in $\mathcal{A}_{n,k}$ to its corresponding k-tree: this will prove that the obtained code is bijective.

The encoding algorithm takes as input a k-tree T_k with n nodes and computes a code in $\mathcal{A}_{n,k}$. It is summarized in the following 4 steps:

Encoding Algorithm

1. Identify Q, the k-clique adjacent to the leaf with maximum label l_M in 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 codeword S the pair corresponding to $\phi(l_M)$.
4. Return the codeword (Q, S).

Assuming that the input k-tree is represented by adjacency lists adj, we now detail how to implement the Encoding Algorithm in linear time.

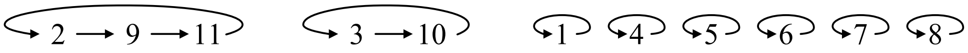


Figure 6.5: Graphical representation of ϕ for 3-tree in Figure 6.1(a).

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 should get labels in $\{n-k+1, n-k+2, \ldots, n\}$. This can be achieved by a relabeling ϕ (i.e., a permutation of labels) defined as follows:

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 $q \in \{n-k+1, \ldots, n\} \setminus Q$, $\phi(q) = i$ such that $\phi^j(i) = q$ and j is maximized.

Figure 6.5 provides a graphical representation of the permutation ϕ corresponding to the 3-tree in Figure 6.1(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.

The Rényi k-tree R_k is obtained relabeling T_k according with 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 6.3(a) gives the Rényi 3-tree R_3 obtained by relabeling the T_3 of Figure 6.1(a) according with ϕ represented in Figure 6.5. The root of R_3 is $\{9, 10, 11\}$.

Let us now prove that the overall running time 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.

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

function COMPUTE-PHI

```
1. for $q_i \in Q$ in increasing order do
2.     $\phi(q_i) = n - k + i$
3. for $i = 1$ to $n - k$ do
4.     $j = i$
5.     while $\phi(j)$ is assigned do
6.         $j = \phi(j)$
7.     $\phi(j) = i$
```

Let us show the correspondence between rules in the definition of the function ϕ and lines of the algorithm: assignments of rule 1 are made by the loop in Line 1. 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 $Q \cup \{n-k+1, \ldots, n\}$. Moreover, for all other nodes the inner while loop scans each permutation cycle only once, according to rule 3 of the definition of ϕ. Thus the program runs in $O(n)$ time.

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

function ORDER-ADJACENCY-LISTS

```
for $i = 1$ to $n$ do
    for each $j \in adj(i)$ do
        append $i$ to $newadj(j)$
return $newadj$
```

Step 2. The goal of this step is to build the characteristic tree T of R_k. In order to guarantee linear running time 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 the running time, the explicit removal of each node should be avoided. We keep this information by marking removed nodes and by 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 6.2). 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 $K_0 = R$).

The algorithm to Prune R_k is detailed below. Its overall running time is $O(nk)$. Indeed, it removes $n-k$ nodes and each removal requires $O(k)$ time.

PRUNE R_k ALGORITHM

function remove(x)

```
let $K_x$ be $adj(x)$ without all marked elements
create a new node in $T$ with label $x$
mark $x$ as removed
for each unmarked $y \in adj(x)$ do
    $d(y) = d(y) - 1$
```

main

```
for $v = 1$ to $n - k$ do
    $w = v$
```

```
  if $d(w) = k$ then
    remove(w)
    while ∃ unmarked $u \in adj(w)$ : $u < v$ and $d(u) = k$ do
      $w = u$
      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 root 0 (see Remark 6.7), 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 such that $K_v = R$. 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:

function ADD-EDGES

```
1. for each $v \in [1, n-k]$ in reversed pruning order do
2.   if $K_v = R$ then
3.     $p(v) = 0$
4.     $level(v) = 1$
5.   else
6.     choose $w \in K_v$ whit maximum $level(w)$
7.     $p(v) = w$
8.     $level(v) = level(w) + 1$
```

This function requires $O(nk)$ time. Indeed, 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. This information can be computed in $O(nk)$ by simply scanning lists K_v. Indeed, the execution of ORDER-ADJACENCY-LISTS at the end of Step 1 ensures that elements in all K_v appear in increasing order.

Figure 6.3(c) shows the characteristic tree computed for the Rényi 3-tree of Figure 6.3(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 codeword S consisting in a list of $n - k - 1$ pairs. For each $v \in \{0, 1, 2, \ldots, n - k\} \setminus \{0, x\}$ there is a pair $(p(v), \ell(v, p(v)))$ taken from the set $\{(0, \varepsilon)\} \cup ([1, n - k] \times [1, k])$. As it is, the obtained codeword S 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 effectively reduce the codeword length, we must guarantee that $\phi(l_M) \neq x$.

Lemma 6.9. *Given a k-tree T_k with n nodes, let l_M be the maximum leaf of T_k and ϕ the permutation described in Step 1. Then, if x is chosen as $\phi(\min\{v \notin Q\})$, it holds $\phi(l_M) \neq x$.*

Proof. From Remark 6.2, 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, $\phi(l_M) \neq \phi(\overline{q}) = x$. □

The removal of the redundant pair from the codeword S completes Step 3. Since the Generalized Dandelion Code can be computed in linear time, the overall running time of the encoding algorithm is $O(nk)$.

It should be 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 codeword length from $n-k-1$ to $n-k-2$: this is crucial to obtain a bijective code for all k-trees. The same could have been done with E-R Bijection, Happy code, and MHappy code as well. Blob code and Prüfer-like codes, can be generalized to encode edge labeled trees, obtaining bijection between Rényi k-trees and codewords in $\mathcal{B}_{n,k}$. However, with these codes, it is not clear how to identify a removable redundant pair. This means that not any code for Rényi k-trees can be directly exploited to obtain a code for k-trees.

The final codeword (Q, S) belongs to $\mathcal{A}_{n,k}$, indeed $Q \in \binom{[1,n]}{k}$ 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 6.3(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 6.3(a). The 3-tree T_3 in Figure 6.1(a) is encoded 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 $\phi(\overline{q}) = 1$.

6.5.2 Decoding Algorithm

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

Decoding Algorithm

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. Since Q is known, it is straightforward to compute $\overline{q} = \min\{v \in [1,n] : v \notin Q\}$ and ϕ as described in Step 1 of the ENCODING ALGORITHM. Since all internal nodes of T explicitly appear in S (see Remark 6.8), 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 deduce 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 now detailed. We assume that each K_v is a list of k integers in increasing order.

REBUILD R_k ALGORITHM

```
1. initialize $R_k$ as the $k$-clique $R$ on $\{n-k+1, n-k+2, \ldots, n\}$
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.     $K_v = K_{p(v)} \setminus \{w\} \cup \{p(v)\}$
8.   add $v$ to $R_k$
9.   add to $R_k$ all edges $(u, v)$ such that $u \in K_v$
```

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 $O(nk)$. The only step of deserves explanation is Line 7 of the REBUILD R_k ALGORITHM. 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 when $K_v = R = \{n-k+1, \ldots, n\}$ it is trivially ordered, all K_v can be easily produced as ordered lists.

6.6 Compact Representation

In this section we present some technical details about the physical representation of codewords in memory. A codeword in $\mathcal{A}_{n,k}$ can be represented efficiently using 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 corresponding to any other pair in $[1, n-k] \times [1, k]$. Otherwise one more bit must be used. 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 $Q \in \binom{n}{k}$.

The easiest form 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 $\binom{n}{k}$.

A non redundant representation of Q is given by its index in the lexicographically ordered list L of all $X \in \binom{n}{k}$. 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 techniques (or with more sophisticate approaches [103]) 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 it is not correct to assume that basic operations require constant time).

6.7 Concluding Remarks

In this chapter we have introduced a new bijective code for labeled k-trees, together with coding and decoding algorithms whose running time is linear with respect to the input size. To the best of our knowledge, no linear time algorithms for encoding and decoding k-tree have been presented in the literature before our work.

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 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 changes, to bijectively encode 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 codeword 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, it is enough to 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 our work completely solves the problem of coding and decoding k-trees efficiently. As a future direction for research in this topic, we propose to work on bijective codes for the class of partial k-trees.

Chapter 7

Counting k-Arch Graphs

The class of k-trees studied in the previous chapter can be further generalized by relaxing the second constraint of Definition 6.1 asking for the node set K to be a clique. Graphs belonging to this class, introduced by Todd [100], are known as the k-arch graphs. Formally a k-arch graph can be defined as follows:

1. A complete graph on k nodes is a k-arch graph.
2. If $A'_k = (V, E)$ is a k-arch graph, $K \subseteq V$ of cardinality k and $v \notin V$, then $A_k = (V \cup \{v\}, E \cup \{(v, x) \mid x \in K\})$ is also a k-arch graph.

An attempt to count the number of labeled k-arch graphs has been made by Lamathe [72]. He used on k-arch graphs the very same generalization of the Prüfer code given by Rényi and Rényi [94] for k-trees (see Section 6.2). Thus each k-arch graph is associated with a strings over the alphabet $\Sigma = \binom{[1,n]}{k}$. He claimed that this correspondence is a bijection and that the number of labeled k-arch graphs on n nodes is:

$$\binom{n}{k}^{n-k-1}$$

Unfortunately this result is not correct, as many codewords do not represent any k-arch graph. We prove the flaw in Lamathe's formula by showing a simple counterexample (in Section 7.2). As a novel contribution of this thesis we provide the characterization of valid codeword and exploit it to define a recursive function that computes the number of labeled k-arch graphs of n nodes, for any given n and k [21].

This chapter is organized as follows: in Section 7.1 we explicitly recall the generalization of the Prüfer code used by Lamathe on k-arch graphs. In Section 7.2 we discuss the decoding procedure and characterize the set of valid codewords. The main counting result is given in Section 7.3.

7.1 Encoding k-Arch Graphs

Let $\mathcal{A}_k^n$ be the set of all k-arch graphs of n nodes and let $\mathcal{B}_k^n$ be the set of all possible strings of length $n-k-1$ over the alphabet $\binom{[1,n]}{k}$. We use the notation $adj(v)$ to identify the set of all nodes adjacent to a given node v, and the term *k-leaf* to mean a node u such that $|adj(u)| = k$; any other node v has $|adj(v)| > k$ and is called *internal.*

Let us define the following function:

$$\rho(A_k^n) = \begin{cases} \varepsilon, & \text{if } A_k^n \text{ is a single } k+1 \text{ clique;} \\ adj(min\{v \in A_k^n : |adj(v)| = k\}) :: \rho(A_k^n \smallsetminus \{v\}), & \text{otherwise.} \end{cases}$$

The function ρ is the injective function between $\mathcal{A}_k^n$ and $\mathcal{B}_k^n$ used by Lamathe [72], i.e., the generalization made by Rényi and Rényi [94] of the Prüfer bijection applied to k-arch graphs. The recursion described by ρ operates a pruning of the k-arch graph A_k^n that starts from the smallest k-leaf v; as v is removed from A_k^n, its adjacent set constitutes the first symbol of the codeword. This symbol is concatenated (by string concatenation operator ::) to the string obtained by recursively applying the function to the pruned graph. The recursion terminates when the pruning gives a clique on $k+1$ nodes, as ρ applied to a clique gives the empty string ε.

Note that, by definition of k-arch graphs, every subgraph produced during the pruning process is a k-arch graph.

It is worth to notice that we are assuming $n > k$, analogously the Prüfer code assumes the tree to have at least 2 nodes. When $n = k$ the only

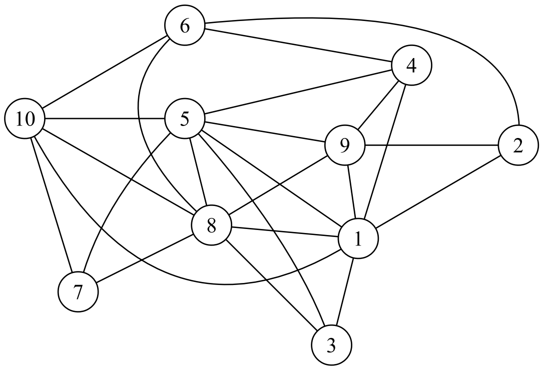


Figure 7.1: A labeled 3-arch graph on 10 nodes.

admissible k-arch graph is a single clique then $|\mathcal{A}_k^k| = 1$, when $n < k$ obviously $|\mathcal{A}_k^n| = 0$.

Let us show an example of the encoding process realized by the function ρ. Starting from the 3-arch graph of Figure 7.1 we prune it by recursively removing the smallest k-leaf. At each step the set of nodes adjacent to the removed k-leaf is added to the codeword.

The smallest k-leaf of the initial graph is $v_1 = 2$ and its adjacent nodes are $B_1 = \{1, 6, 9\}$. Then node 2 is removed from the graph and the smallest k-leaf in the resulting graph is $v_2 = 3$ implying $B_2 = \{1, 5, 8\}$. Iterating this procedure we obtain $v_3 = 6$, $v_4 = 4$, $v_5 = 7$, $v_6 = 9$ and $B_3 = \{4, 8, 10\}$, $B_4 = \{1, 5, 9\}$, $B_5 = \{5, 8, 10\}$, $B_6 = \{1, 5, 8\}$ respectively. The remaining graph is a single clique of 4 nodes $\{1, 5, 8, 10\}$. Therefore the resulting codeword is $(B_1, B_2, B_3, B_4, B_5, B_6) = (\{1, 6, 9\}, \{1, 5, 8\}, \{4, 8, 10\}, \{1, 5, 9\}, \{5, 8, 10\}, \{1, 5, 8\})$.

For a given k-arch graph A_k^n, we say a node $v \in V(A_k^n)$ *appears* in $\rho(A_k^n)$ if there exists $B_i \in \rho(A_k^n)$ such that $v \in B_i$.

Lemma 7.1. *v is an internal node in A_k^n if and only if it appears in $\rho(A_k^n)$.*

Proof. Consider an internal node v in A_k^n: its initial degree is strictly greater than k. The pruning process operated by ρ ends with a $(k+1)$-clique, where each node has degree k: either v has been eliminated in some step or it belongs to the remaining clique; in both cases its degree must decrease to k.

Since the degree of an internal node v can decrease only if in some step i a k-leaf adjacent to v is removed, v must belong to B_i.

Let us now show that if an element appears in $\rho(A_k^n)$, then it is an internal node. Consider a k-leaf v, and suppose by contradiction that there exists some value i such that $v \in B_i$. This means that after removing a k-leaf on step i, in the resulting graph node v has degree $k-1$. This contradicts the fact that each subgraph produced during the encoding process is k-arch graph. □

Lemma 7.2. *Function ρ is injective.*

Proof. We have to show that, given two k-arch graphs $A_k^{n\prime}$ and $A_k^{n\prime\prime}$, if $\rho(A_k^{n\prime}) = \rho(A_k^{n\prime\prime}) = (B_1, \ldots, B_{n-k-1})$ then $A_k^{n\prime} = A_k^{n\prime\prime}$.

Let us proceed by induction on $n-k$. If $n-k=1$, $\rho(A_k^{n\prime}) = \rho(A_k^{n\prime\prime}) = \varepsilon$, then $A_k^{n\prime} = A_k^{n\prime\prime}$ as the only k-arch graph on $k+1$ nodes is a $(k+1)$-clique.

For inductive hypothesis, assume the thesis holds when $n-k<h$. We have to prove that it holds when $n-k=h$.

In order to have $\rho(A_k^{n\prime}) = \rho(A_k^{n\prime\prime})$, for Lemma 7.1, the sets of internal nodes and the sets of k-leaves in $A_k^{n\prime}$ and $A_k^{n\prime\prime}$ must coincide. It follows that the minimum k-leaf v_1 in $A_k^{n\prime}$ coincides with the minimum k-leaf in $A_k^{n\prime\prime}$ and both are adjacent to the same node set B_1. Moreover, the graphs obtained by pruning v_1 from $A_k^{n\prime}$ and $A_k^{n\prime\prime}$, in order to produce the same substring $(B_2, \ldots, B_{n-k-1})$, have to be the same graph by inductive hypothesis. This implies $A_k^{n\prime} = A_k^{n\prime\prime}$, as removing the same node and the same edge set from them we obtain the same graph. □

7.2 Decoding k-Arch Graphs

In this section we show how to revert function ρ in order to rebuild an encoded k-arch graph.

Starting from a codeword $(B_1, \ldots, B_l)$ that is the encoding of an unknown k-arch graph A_k^n, initially we need to recover values n and k: $k = |B_1| =$

$|B_2| = \cdots = |B_l|$ and, since $l = n - k - 1$, we can derive $n = l + k + 1$. The node set of A_k^n is $[1, n]$ so, to complete the decoding process, we need to reconstruct its edge set.

In view of Lemma 7.1 it is easy to derive the set of all k-leaves of A_k^n as $[1, n] \setminus \bigcup B_i$. We can compute v_1 (the first k-leaf removed during the encoding process) as the minimum of this set. We also know $adj(v_1) = B_1$.

Now, v_2 is the smallest k-leaf of $A_k^n \setminus \{v_1\}$ and we know both the node set of this k-arch graph (i.e., $[1, n] \setminus \{v_1\}$) and its codeword $(B_2, \ldots, B_l)$. Then $v_2 = \min\{v \in [1, n] \setminus \{v_1\} \setminus \bigcup_{i=2}^{l} B_i\}$.

Generalizing this idea it is possible to derive a formula analogous to the one given by Prüfer for trees:

$$v_i = \min \left\{ v \in [1, n] \setminus \{v_h\}_{h<i} \setminus \bigcup_{j \geq i} B_j \right\} \qquad \forall i \in [1, l] \tag{7.1}$$

Knowing the k-leaf removed at each step of the encoding process it is easy to rebuild the edge set of A_k^n. Indeed, all the $k + 1$ nodes not in $\{v_1, \ldots, v_l\}$ form a clique and each v_i should be connected with all nodes in B_i. We will refer to this decoding process as ρ^{-1}. It is easy to see that the codomain of ρ^{-1} is $\mathcal{A}_k^n$.

Not all strings in $\mathcal{B}_k^n$ are eligible for this decoding procedure. Indeed, ρ^{-1} implicitly requires that, at each step i, the set from which each v_i is chosen (Equation 7.1) must be not empty. We now show a simple string not corresponding to the encoding of any k-arch graph: this is the easiest counterexample that proves the incorrectness of Lamathe's formula.

Consider the string $(\{1, 2\}, \{3, 4\}, \{5, 6\})$: in this case $k = 2$ and $n = 3 + 2 + 1 = 6$. Since the set $[1, 6] \setminus (\{1, 2\} \cup \{3, 4\} \cup \{5, 6\})$ is empty, there is no value for v_1, so there can not exist any k-arch graph whose encoding is $(\{1, 2\}, \{3, 4\}, \{5, 6\})$.

It is quite easy to see, from definition of ρ^{-1}, that $\rho^{-1}(\rho(A_k^n)) = A_k^n$ for each k-arch graph A_k^n. We now characterize all those strings in $\mathcal{B}_k^n$ resulting by the encoding of some k-arch graph. Let us call the set of these strings

$\mathcal{C}_k^n \subseteq \mathcal{B}_k^n$. Notice that $\mathcal{C}_k^n$ is the image of $\mathcal{A}_k^n$ under function ρ, i.e., $\mathcal{C}_k^n = \rho(\mathcal{A}_k^n)$.

Theorem 7.3. *Given $(B_1, \ldots, B_l) \in \mathcal{B}_k^n$ if $\exists\{v_1, \ldots, v_l\} \in \binom{[1,n]}{l}$ such that $v_i \notin \bigcup_{j=i}^{l} B_j$ then $(B_1, \ldots, B_l) \in \mathcal{C}_k^n$.*

Proof. The existence of such a sequence $\{v_1, \ldots, v_l\} \in \binom{[1,n]}{l}$ ensures that the decoding process can be applied successfully, but this is not enough to ensure $(B_1, \ldots, B_l) \in \mathcal{C}_k^n$. Indeed there is a reasonable doubt that the k-arch graph $A_k^n = \rho^{-1}(B_1, \ldots, B_l)$ obtained by decoding an arbitrary string in $\mathcal{B}_k^n$, can produce a different string $(B_1', \ldots, B_l') = \rho(A_k^n)$ when encoded, thus implying $(B_1, \ldots, B_l) \notin \mathcal{C}_k^n$. We will show this is not the case.

Without loss of generality assume that $v_1, \ldots, v_l$ coincides with the sequence of nodes chosen by ρ^{-1} at each step during the decoding process. Now, by induction on l, we prove that $\rho(\rho^{-1}(B_1, \ldots, B_l)) = (B_1, \ldots, B_l)$.

When $l = 0$, the string can only be ε, the resulting graph is a $(k+1)$-clique and its encoding is again ε. We assume, by inductive hypothesis, the thesis holds for any string of length $l < h$ and we prove it holds for strings of length $l = h$. First note that if the string $(B_1, \ldots, B_l)$ is decoded as the k-arch graph A_k^n, then the substring $B_2, \ldots, B_l$ is decodable and results in the graph $A_k^{n-1} = A_k^n \setminus \{v_1\}$. By inductive hypothesis $\rho(A_k^{n-1}) = (B_2, \ldots, B_l)$ (here the node set does not contain v_1). The degree of v_1 in A_k^n is $|B_1| = k$, so it is a k-leaf. Any other node with label smaller than v_1 appears in $(B_1, \ldots, B_l)$, as otherwise ρ^{-1} would have done a different choice for v_1. This implies that v_1 is the minimum k-leaf in A_k^n. Then $\rho(A_k^n) = adj(v_1) :: \rho(A_k^{n-1}) = (B_1, \ldots, B_l)$. □

Since in the proof of Theorem 7.3 we proved that $\rho(\rho^{-1}(B_1, \ldots, B_l)) = (B_1, \ldots,\ B_l)$ for each codeword in $\mathcal{C}_k^n$, we can state that $\rho^{-1} : \mathcal{C}_k^n \to \mathcal{A}_k^n$ is exactly the inverse function of $\rho : \mathcal{A}_k^n \to \mathcal{C}_k^n$.

7.3 Enumerating k-Arch Graphs

We are interested in finding the number of k-arch graphs on n nodes, i.e., $|\mathcal{A}_k^n|$. Since $|\mathcal{A}_k^n| = |\mathcal{C}_k^n|$, in order to count labeled k-arch graphs we will count

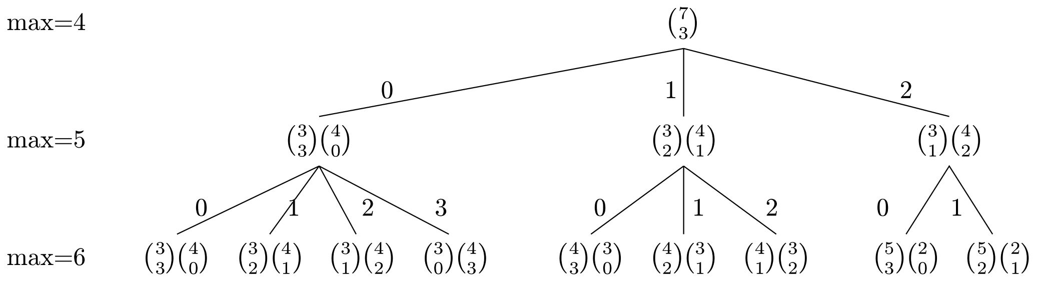

max=4
$\binom{7}{3}$
0
1
2
max=5
$\binom{3}{3}\binom{4}{0}$
$\binom{3}{2}\binom{4}{1}$
$\binom{3}{1}\binom{4}{2}$
0
1
2
3
0
1
2
0
1
max=6
$\binom{3}{3}\binom{4}{0}$
$\binom{3}{2}\binom{4}{1}$
$\binom{3}{1}\binom{4}{2}$
$\binom{3}{0}\binom{4}{3}$
$\binom{4}{3}\binom{3}{0}$
$\binom{4}{2}\binom{3}{1}$
$\binom{4}{1}\binom{3}{2}$
$\binom{5}{3}\binom{2}{0}$
$\binom{5}{2}\binom{2}{1}$

Figure 7.2: Recursion tree for counting 3-arch graphs on 7 nodes.

valid codewords. The condition for a string $(B_1, \ldots, B_l)$ to be a valid codeword of a k-arch graph (stated in Theorem 7.3) can be easily reformulated as:

$$\forall i : 1 \leq i \leq l, \quad |\bigcup_{h=i}^{l} B_h| \leq n - i \tag{7.2}$$

Exploiting Equation 7.2, it possible to define a recursive function to count the number of labeled k-arch graphs on n nodes. Before providing this general formula, let us show an example of our approach applied to $|\mathcal{C}_3^7|$.

The basic idea is to simulate the generation of a valid codeword (B_1, B_2, B_3), from right to left, and count how many choices we have at each step. The choice for B_3 gives $\binom{7}{3}$ alternatives, as Equation 7.2 requires that no more than 4 different numbers appear in substring (B_3); this substring always contains 3 distinct numbers, then the requirement is always satisfied.

Now consider B_2. Equation 7.2 requires at most 5 distinct numbers to appear in substring (B_2, B_3), thus imposing some limits on the choices for B_2. In fact valid choices are those selecting 3, 2 or 1 numbers appearing in B_3 and respectively 0, 1 or 2 unused numbers, giving $\binom{3}{3}\binom{4}{0}$, $\binom{3}{2}\binom{4}{1}$ and $\binom{3}{1}\binom{4}{2}$ distinct alternatives. Similar arguments hold for B_1 and constraints depend on how many distinct numbers appear in (B_2, B_3). More explicitly, since Equation 7.2 imposes to have at most 6 distinct numbers, if u distinct numbers appear in (B_2, B_3), then B_1 can introduce up to $\min(3, 6-u)$ unused numbers.

Figure 7.2 gives the complete recursion tree for the described process. The root represents choices for B_3; children of the root represent choices for

B_2 and leaves represent choices for B_1. For each level, on the left the bound given by Equation 7.2 is reported; labels on edges represent how many new numbers are introduced. $|\mathcal{C}_3^7| = 34405$ is given by the sum of the products of labels given by each leaf-to-root path in the tree:

$$\begin{aligned}\binom{7}{3}\Bigg(\; & \binom{3}{3}\binom{4}{0}\left(\binom{3}{3}\binom{4}{0}+\binom{3}{2}\binom{4}{1}+\binom{3}{1}\binom{4}{2}+\binom{3}{0}\binom{4}{3}\right)+\\ & \binom{3}{2}\binom{4}{1}\left(\binom{4}{3}\binom{3}{0}+\binom{4}{2}\binom{3}{1}+\binom{4}{1}\binom{3}{2}\right)+\\ & \binom{3}{1}\binom{4}{2}\left(\binom{5}{3}\binom{2}{0}+\binom{5}{2}\binom{2}{1}\right)\;\Bigg)\end{aligned}$$

Now we introduce our main result on k-arch graphs.

Theorem 7.4. *The number of labeled k-arch graphs on $n > k+1$ nodes is $|\mathcal{A}_k^n| = f_k^n(n-k-1, 0, k)$ where f_k^n is the recursive function defined as:*

$$f_k^n(i,u,j) = \begin{cases} \binom{n-u}{j}\binom{u}{k-j}, & \text{if } i = 1;\\ \binom{n-u}{j}\binom{u}{k-j}\displaystyle\sum_{c=0}^{\min(k,n-(i-1)-(u+j))} f_k^n(i-1, u+j, c), & \text{otherwise.}\end{cases}$$

When $n = k$ or $n = k+1$ we have $|\mathcal{A}_k^n| = 1$; when $n < k$ we have $|\mathcal{A}_k^n| = 0$.

Proof. Given the string $(B_1, \ldots, B_l) \in \mathcal{C}_k^n$, we call *characteristic* of this string the vector $\overline{c} = (c_1, \ldots, c_{l-1})$ such that $c_i = |B_i \setminus \bigcup_{j>i} B_j|$, i.e., the number of elements in B_i that do not appear in the substring $(B_{i+1}, \ldots, B_l)$.

Consider the recursion tree generated by applying the function f_k^n to $(n-k-1, 0, k)$. This tree is a generalization of the one presented in Fig. 7.2 for the special case $n = 7$ and $k = 3$: node labels correspond to the binomials product and edge labels correspond to the value of the variable c discriminating recursive applications of function f_k^n.

Notice that, considering the edge labels in any leaf to root path of this tree, we obtain a vector $(c_1, \ldots, c_{n-k-2})$ which represents the sequence of newly inserted numbers (from right to left), and so it coincides with the characteristic of some string in $\mathcal{C}_k^n$. It is also true that if $\overline{c}$ is the characteristic of a string in $\mathcal{C}_k^n$, then a leaf to root path whose edge labels vector is $\overline{c}$ must exist.

$|\mathcal{C}_k^n|$ can be obtained by summing cardinalities of disjoint sets of strings sharing the same characteristic. The size of any such set is given by the product of node labels following the corresponding leaf to root path in the recursion tree. By summing those products, we thus obtain $|\mathcal{C}_k^n|$, i.e., the value computed by $f_k^n(n-k-1, 0, k)$. □

7.3.1 Experimental Results

We implemented the recursive function to enumerate the labeled k-arch graphs on n nodes using the open source algebraic system *PARI/GP* [31]. The code performing the counting is given in Figure 7.3.

```
f(n,k,i,u,j)={
    local(s=0);
    if (i==1,
        binomial(n-u,j)*binomial(u,k-j),
        for (c=0, min(k,n-(i-1)-(u+j)),
            s+=f(n,k,i-1,u+j,c)
        );
        binomial(n-u,j)*binomial(u,k-j)*s
    )
}
```

Figure 7.3: PARI/GP code implementing the recursive function f_k^n.

The size of the recursion tree is exponential in the order of $(k+1)^{n-k-2}$ so the value can only be computed if the difference between n and k is small. As done by Lamathe we report values of $|\mathcal{A}_k^n|$ for $n \in [1, 10]$ and $k \in [1, 7]$ in the following table:

$k\backslash n$	1	2	3	4	5	6	7	8	9	10
1	1	1	3	16	125	1296	16807	262144	4782969	100000000
2	0	1	1	6	100	3285	177471	14188888	1569185136	229087571625
3	0	0	1	1	10	380	34405	5919536	1709074584	764754595200
4	0	0	0	1	1	15	1085	216230	92550276	74358276300
5	0	0	0	0	1	1	21	2576	982926	898027452
6	0	0	0	0	0	1	1	28	5376	3568950
7	0	0	0	0	0	0	1	1	36	10200

The first row of this table gives exactly the well known Cayley's formula, as 1-arch graphs are trees. Apart from this row (reported as Sequence A000272) no other row of the table was listed in the on-line Encyclopedia of Integer Sequences [98] before our work.

7.4 Concluding Remarks

In this chapter we have presented a recursive function that computes the number of labeled k-arch graphs of n nodes, for any given n and k. In order to obtain this function, we have used a code that maps labeled k-arch graphs to strings and we have derived the counting function by characterizing valid code strings. Moreover, we have proved the counting function for k-arch graphs provided by Lamathe to be incorrect by showing a counterexample.

Unfortunately this result does not help us in developing a bijective code for k-arch graphs. It remains an open problem to find, provided that it exists, a closed formula for the number of k-arch graphs $|\mathcal{A}_k^n|$, when $k > 1$. When $k = 1$, from Cayley's formula, we have $|\mathcal{A}_1^n| = n^{n-2}$. Furthermore, it would be interesting to investigate k-arch graphs with fixed or arbitrary root.

Chapter 8

Informative Labeling Scheme for LCA on Dynamic Trees

In this chapter we abandon the idea of labels as simple unique identifiers of nodes, as seen in all previous chapters. Here we investigate a richer concept of label that makes it possible to perform computations directly from node labels: Informative Labeling Scheme (ILS). We focus on ILS for trees. As the idea of ILS naturally realizes a localization of the information required to perform a computation, we decided to exploit this concept to design concurrent data structures. As a first example we focus on the lowest common ancestor (lca) problem for dynamic trees.

Namely, our goal is to associate each node of a tree with a label such that it will be possible to compute the lowest common ancestor of any two nodes directly from their labels. Moreover it should be reasonably efficient to update the labels in order to reflect changes in the tree. We will also use locking system to allow multiple processors to access the data structure simultaneously.

In the second part of this chapter we will present an experimental comparison between our data structure and an ILS for lowest common ancestor queries introduced by Peleg for static trees [87].

This chapter is organized as follows: initially we recall the concept of Informative Labeling Scheme and describe the concurrency model we consider for our data structure. Then we recall the ILS introduced by Peleg for answering lca queries. In Section 8.3 we introduce our ILS for lca on dynamic trees while in Section 8.4 we describe how to use it in a concurrent setting. Finally, in Section 8.5, we present our experimental comparison.

8.1 Preliminaries

An Informative Labeling Scheme (ILS) [87, 88] for a target function f is formally defined as a couple of algorithms $(\mathcal{M}, \mathcal{D})$. The Marker Algorithm $\mathcal{M}$, given a graph G, computes a $label(v)$, for each node v in G. The Decoding Algorithm $\mathcal{D}$ is then used to compute, for each pair of nodes u, v in G the target function $f(u, v)$ using only $label(u)$ and $label(v)$. In other words $\mathcal{D}(label(u), label(v)) = f(u, v)$. The primary goal of a labeling scheme is to minimize the maximum label length, while keeping queries fast.

Adjacency labeling schemes were first introduced by Breuer and Folkman in [14, 15], and further studied in [63]. The interest in informative labeling schemes, however, was revived only more recently, after Peleg showed the feasibility of the design of efficient labeling schemes capturing distance information [86]. Since then, upper and lower bounds for labeling schemes have been proved on a variety of graph families (including weighted trees, bounded arboricity graphs, intersection-based and c-decomposable graphs) and for a large variety of queries, including distance [2, 30, 49, 50, 64, 67, 99], tree ancestry [1, 3], flow and connectivity [66].

We focus on labeling schemes for answering least common ancestor queries in trees. We recall that the least common ancestor of any two tree nodes u and v, denoted as $lca(u, v)$, is the common ancestor of u and v having the smallest distance to u (and to v). The least common ancestor problem has been extensively studied over the last three decades in different models of computation [4, 7, 33, 34, 58, 97]. Labeling schemes for least common ancestors are mainly useful in routing messages on tree networks: the ability to compute the identifier of the least common ancestor of any two nodes u and v turns out to be useful when a message has to be sent from u to v in the network, because the message has to go through $lca(u, v)$. Other applications are related to query processing in XML search engines and distributed computing in peer-to-peer networks (see, e.g., [3, 11, 65]). In [87], Peleg has proved that for trees there exists a labeling scheme for least common ancestors using $\Theta(\log^2 n)$-bit labels, which is also shown to be asymptotically optimal (as usual n represents the number of nodes in the tree).

In spite of a large body of theoretical works, to the best of our knowledge only few experimental investigations of the efficiency of compact labeling schemes have been addressed in the literature [30, 65]. For this reason we decided to enrich our study with an experimental comparison (see Section 8.5).

As stated in the introduction of this chapter we exploit ILS to design a concurrent data structure for lca on dynamic trees. The tree is dynamic in the sense that we allow it to grow by insertion of new leaves.

Our data structure will implement two operations:

Query: given any two tree nodes u and v, compute their lowest common ancestor $lca(u, v)$;

Update: given a tree node p, add to the tree a new node v as a child of p.

8.1.1 The Concurrency Model

Throughout this chapter we focus on a multiprocessor system in which processors communicate by writing and reading shared variables in a common memory address space. We assume that processors work asynchronously and that each processor has its own local (i.e., non shared) memory. Moreover, in order to use timestamps associated to shared variables, we assume that processors share a common clock: this is typically not a restrictive assumption in a multiprocessor system.

We will use locking primitives to guarantee concurrent access to our data structures. In particular, we will analyze our data structures under the concurrent read, exclusive write (CREW) model by using two different kinds of locks: *shared* and *exclusive*. Several processes can hold shared locks on a variable simultaneously, whereas if one process holds an exclusive lock, then no other process may hold any lock on that variable.

For more information about concurrent data structures, locking primitives and related matters we refer the interested reader to [76].

We remark that concurrent data structures are much more difficult to design and analyze than sequential ones: indeed, in the design of a concurrent data structure, the goal is to allow concurrent operations to proceed in parallel when they do not access the same parts of the data structure. On the other side, the presence of locks introduces a sequential bottleneck on the execution of the operations, thus decreasing the speedup. This bottleneck can be reduced by using a fine-grained locking scheme such that multiple locks of small granularity can be introduced to protect different parts of the data structure.

Correctness analysis. In order to prove that a data structure is correct under concurrent operations, we will show that in every execution there exists a total ordering of the operations with the following properties: (1) the ordering is consistent with the desired insert/search semantics, and (2) if one operation completes before another begins, then the first operation precedes the second one in the ordering.

8.2 Peleg's Labeling Scheme

In [87], Peleg has proved that for trees there exists a labeling scheme for least common ancestors using $\Theta(\log^2 n)$-bit labels. Peleg's labeling scheme hinges upon two main ingredients: a decomposition of the tree into paths, and a suitable encoding of information related to such paths into the node labels. Peleg's data structure uses an *ad hoc* path decomposition as well as an *ad hoc* label structure.

Let T be a tree with n nodes rooted at a given node r. As usual, for any node u we denote its parent and its level in T by $p(u)$ and $l(u)$, respectively (the root has level 0). The tree is decomposed into a set of node disjoint paths, that we will call *solid paths.* For any solid path π, we denote by

$head(\pi)$ the node of π with smallest level. We will also say that a solid path π is an *ancestral solid path* of a node u if $head(\pi)$ is an ancestor of u.

Decomposition by Large Child. This decomposition hinges upon the distinction between small and large nodes: a nonroot node v with parent u is called *small* if $|T_v| \leq |T_u|/2$, i.e., if its subtree contains at most half the number of nodes contained in its parent's subtree. Otherwise, v is called *large*. It is not difficult to see that any node has at most one large child: we will consider the edge to that large child, if any, as a solid edge. Solid edges induce a decomposition of the tree into solid paths: we remark that the head of any solid path π is always a small node, while all the other nodes in π must be large. Each node can have at most $\lceil \log n \rceil$ small ancestors, and thus at most $\lceil \log n \rceil$ ancestral solid paths (unless otherwise stated, all logarithms will be to the base 2).

The Marker Algorithm $\mathcal{M}_p$. The Peleg labeling scheme is based on a depth-first numbering of the tree T: as a preprocessing step, each node v is assigned an interval $Int(v) = [DFS(v); DFS(w)]$, where w is the last descendent of v visited by the depth-first tour and $DFS(x)$ denotes the depth-first number of node x. The label of each node v of the tree is defined as follows:

$$label(v) = < Int(v), list(v) >$$

where $list(v)$ contains information related to all the heads $(t_1, t_2, \ldots, t_h)$ of solid paths from the root of T to v: for each head t_i, $list(v)$ contains a quadruple $(t_i, l(t_i), p(t_i), succ_v(t_i))$, where $succ_v(t_i)$ is the unique child of t_i on the path to node v. We remark that this is slightly different (and optimized) with respect to the scheme originally proposed in [87].

The Decoder Algorithm $\mathcal{D}_p$. We now describe how to query for a lowest common ancestor: given two nodes u and v, the algorithm infers their least common ancestor $z = lca(u, v)$ using only information contained in $label(u)$ and $label(v)$. By well-known properties of depth-first search, we have that for every two nodes x and y of T, $Int(x) \subseteq Int(y)$ if and only if x is a descendent of y in T: this fact can be easily exploited to check whether the least common ancestor z coincides with any of the two input nodes u and v. If this is not the case, let $(u_1, u_2, \ldots, u_h)$ and $(v_1, v_2, \ldots, v_k)$ be the heads of solid paths from the root of T to u and v, respectively: information about these heads is maintained in the node labels. The algorithm finds the least common ancestor head h, which is identified by the maximum index i such that $u_i = v_i$. If $succ_u(h) \neq succ_v(h)$, then h must be the least common ancestor. Otherwise, the algorithm takes the node of minimum level between u_{i+1} and v_{i+1}, and returns its parent as the least common ancestor. We refer to [87] for a formal proof of correctness. Here, we limit to remark that both depth-first numbering and information about successors appear to be crucial in this algorithm.

8.3 A Dynamic Sequential Data Structure

The Peleg's labeling scheme is not suitable for dynamic trees, indeed it hinges upon the depth-first numbering of the tree nodes. The insertion of a new node can affect the depth-first numbering of many other nodes. This implies that at each insertion a relevant part of the data structure needs to be recomputed.

Our data structure relays on the concept of ILS and, similarly to the Peleg's one, is based on a decomposition of the tree into solid path. The decomposition we use is the same used in [71] and is described below. Moreover, in order to avoid depth-first numbering, we maintain slightly different information in node labels.

We first present a sequential data structure for maintaining lowest common ancestors upon insertions of new tree nodes and we discuss its correctness. Although not optimal in a sequential setting, this data structure is well suited for an efficient concurrent implementation, as we will show in Section 8.4.

8.3.1 Tree Decomposition

For each edge (u, v), we call the edge *solid* if and only if $\lceil \log |T_u| \rceil = \lceil \log |T_v| \rceil$. We will also say that v is a *solid child* of u. It is not difficult to see that solid edges decompose T into *solid paths*, also known as centroid paths [33]. We will say that a solid path π has *rank* i if, for all nodes v belonging to π, the size of the subtree rooted at v satisfies the following inequality:

$$2^i \leq |T_v| < 2^{i+1} \tag{8.1}$$

solid paths univocally partition the tree into disjoint paths, as proved in the following lemma:

Lemma 8.1. *For any node u there exists at most one child v such that (u, v) is solid.*

Proof. Assume by contradiction that both (u, v_1) and (u, v_2) are solid, where v_1 and v_2 are distinct children of u such that, without loss of generality, $|T_{v_2}| \geq |T_{v_1}|$. Then it must be $\lceil \log |T_{v_1}| \rceil = \lceil \log |T_{v_2}| \rceil$. Since $|T_u| \geq |T_{v_1}| + |T_{v_2}| \geq 2|T_{v_1}|$, we have $\lceil \log |T_u| \rceil \geq \lceil \log |T_{v_1}| \rceil + 1$. This contradicts the hypothesis that (u, v_1) is solid. □

We will refer to the partition induced by solid paths as *solid path decomposition*. For any solid path π, we denote by *head*(π) and *tail*(π) the node of π with smallest and largest level, respectively. The root of the tree is always head of a solid path. Notice that a solid path can have length 0 (i.e., it can consist of a single node) and that all the leaves are heads of solid paths. We will say that a solid path π is an *ancestral solid path* of a node u if *head*(π) is an ancestor of u, and that π is an ancestral solid path of a path π' if *head*(π) is an ancestor of *head*(π'). The following property easily follows from Equation 8.1:

Property 8.2. *Each node u has at most $\lceil \log n \rceil$ ancestral solid paths.*

Similarly to [71], we now introduce *relaxed solid paths*: these paths will be useful to deal with dynamic trees in order to avoid frequent recomputations of the decomposition upon insertions of new nodes. We allow a node v to belong to a relaxed solid path of rank i, with $i \geq 2$, if the size of the subtree rooted at v satisfies the following inequality:

$$2^i \leq |T_v| < 2^{i+1}(1+\alpha) \tag{8.2}$$

where α is an arbitrary constant such that $0 < \alpha < 1$. We will call a node *stable* (with respect to rank i) if it satisfies Inequality (8.1), and *unstable* (with respect to rank i) if it satisfies Inequality (8.2) but not Inequality (8.1).

We remark that relaxed solid paths do not univocally induce a partition of the tree. Indeed, it may be the case that an unstable node u in a path π of rank i has two children that satisfy Inequality (8.2) with respect to i: both of these children could therefore belong to π, that would be no longer a path. We will break these ties by including in π at most one of these two children, and by considering the other child as head of a different path π' also of rank i. Namely, if only one of the two children is stable, we choose it as head of π'. Otherwise, we break the tie arbitrarily. We remark that u might also have more than two children: however, since it belongs to a path of rank i, at most two of its children can have size large enough to satisfy Inequality (8.2).

This tie breaking approach, however, introduces a different problem: more than one path of rank i may now appear in the path from the root to any given node, thus invalidating Property 8.2. We will now show that this is not a real problem, since the number of ancestral (relaxed) solid paths of any node can at most double.

Lemma 8.3. *Let u be a node in a relaxed solid path of rank i, and let v and w be two children of u both satisfying Inequality (8.2) with respect to i. Then u and at most one between v and w is unstable with respect to i.*

Proof. We first notice that u satisfies Inequality (8.2) because it belongs to a relaxed solid path of rank i. Since v and w satisfy Inequality (8.2) by hypothesis, we have $|T_u| > |T_v| + |T_w| \geq 2^{i+1}$. Hence, u is unstable. We now consider v and w. If both of them are unstable, then their subtree sizes would be larger than or equal to 2^{i+1} and it would be $|T_u| > 2^{i+2}$. This contradicts the hypothesis that u belongs to a relaxed solid path of rank i. □

Corollary 8.4. *Let π and π' be any two solid paths of rank i such that π is an ancestral path of π'. Then no other solid path can exist between π and π', i.e., no other solid path π'' such that π is an ancestral path of π'' and π'' is an ancestral path of π' can exist.*

Proof. Let us assume by contradiction that there exists a path π'' such that π is an ancestral path of π'' and π'' is an ancestral path of π'. Let v be the head of π'' and let u be its parent. It is easy to see that it must be $rank(\pi) \geq rank(\pi'') \geq rank(\pi')$. This implies that $rank(\pi'') = i$ and that u belongs to a path of rank i. In the following, without loss of generality we will assume that the solid path of node u coincides with π.

Since $rank(\pi) = i = rank(\pi'')$, then π'' must be the result of an application of the tie breaking rule. By Lemma 8.3, u is unstable. Let w be the child of u in π. Lemma 8.3 guarantees that, if w is unstable, then v must be stable. If w is stable, then u cannot be unstable: if this is not the case, then the tie breaking rule would have included v in π instead of w. In both cases, v is stable. This implies that no tie breaking can take place below it, contradicting the existence of π'. □

By Corollary 8.4, we can have at most two consecutive paths with the same rank. Hence, the tie breaking rule implies that the number of (relaxed) ancestral solid paths of any node can at most double, i.e.:

Property 8.5. *Each node u has at most $2\lceil \log n \rceil$ ancestral relaxed solid paths.*

We remark that in [71] the same problem related to relaxed solid paths has been encountered and solved in a different way: our solution is simpler and still allows us to design an ILS whose labels size is $O(\log^2 n)$ bits.

8.3.2 The Data Structure

For each node v of a tree T, we maintain its parent, the list of all its children, and a pointer to its solid child (if any). We also maintain a label defined as follows:

$$label(v) = < isHead(v), list(v) >$$

where $isHead(v)$ is a Boolean value discriminating whether v is the head of its relaxed solid path or not, and $list(v)$ contains information related to all the heads $(t_1, t_2, \ldots, t_h)$ of relaxed solid paths from the root of T to v. Namely, $list(v)$ consists of a sequence of triples:

$$list(v) = [\,(t_1, l(t_1), p(t_1)),\ \ldots,\ (t_h, l(t_h), p(t_h)),\ (v, l(v), p(v))\,]$$

where t_1 always coincides with the root of T. The sentinel triple $(v, l(v), p(v))$ is not necessary when v is head of its path, since in this case $t_h = v$.

For each v head of a relaxed solid path π, we also explicitly maintain in our data structure the size of its subtree $|T_v|$ and the rank of its path $rank(\pi)$.

We now describe the implementation of the update and query operations.

Update. Each time a new node v is added to T as a child of a node u, besides updating the children list of u, we compute $label(v)$ as follows:

$$label(v) = \begin{cases} < true, list(u) :: (v, l(u)+1, u) > & \text{if } isHead(u) \\ < true, list(u) \setminus (u, l(u), p(u)) :: (v, l(u)+1, u) > & \text{otherwise} \end{cases}$$

where :: concatenates a new triple to a list and $\setminus$ is used to remove the sentinel triple of $list(u)$ when u is not head of its solid path. We remark that the new leaf v is always head of a solid path of rank 1, since the relaxed Inequality (8.2) only applies to paths of rank at least 2.

Due to the addition of node v, the subtree size of its ancestors changes, and some ancestor may no longer satisfy Inequality (8.2). This implies that the decomposition into relaxed solid paths and the labels of some nodes may need to be updated. We now describe how to tackle this problem.

After $label(v)$ has been computed, the size $|T_{t_i}|$ of the subtree rooted at t_i is incremented by one for each head t_i in $list(v)$. Let t be the topmost head that no longer satisfies Inequality (8.2), if any. The idea is to move t upward in the decomposition so that, after the update is completed, it will belong to a relaxed solid path of rank $i = \lceil \log |T_t| \rceil$. At this aim, let π' be the relaxed solid path above t: notice that $p(t)$ belongs to π'. We can easily find the head of π' at the end of $list(t)$. By accessing the data associated to $head(\pi')$, we can also retrieve $rank(\pi')$. Three cases may now arise:

1. If $rank(\pi') > i$, then t becomes the head of a new solid path of rank i.
2. If $rank(\pi') = i$ and $p(t)$ has no solid child, then t can join π'.
3. If $rank(\pi') = i$ and $p(t)$ already has a solid child, this prevents t from joining π' and t will be the head of a new relaxed solid path of rank i.

We notice that the second case is a correct implementation of the tie breaking rule described in Section 8.3.1: indeed, t is stable with respect to rank $i = \lceil \log |T_t| \rceil$.

After t has been correctly placed in a relaxed solid path, we recompute from scratch all the information related to t and to its subtree: the labels, the size of the subtree and the rank of each head in T_t, and the decomposition into solid paths. While recomputing the decomposition, we force each node to respect Inequality (8.1); this guarantees that all the unstable nodes that were in the same path of node t before the update will be moved upward in the decomposition together with t. We defer to Section 8.3.3 the proof of correctness of this approach. We will also show that each update requires $O(\log^2 n)$ amortized running time.

We remark that, considering our data structure as an ILS, this update procedure plays the role of the Marker algorithm. The main difference is that a Marker Algorithm is ran only once, as a single precomputation step, while the update takes place each time a new node is inserted in the tree.

Query. Our query algorithm computes the lowest common ancestor $lca(u, v)$ of any two nodes u and v as follows (t_i^u and t_i^v denote the i-th head in $list(u)$ and $list(v)$, respectively):

Lines 1 and 2 consider some trivial cases: either $u = v$ or one of them coincides with the root. In line 3 the algorithm finds the lowest head t_k which

Program 1 Query(u,v)

```
1. if $u = v$ then return $u$
2. if $u = r$ or $v = r$ then return $r$
3. Let $k = \max\{i : t_i^u = t_i^v\}$ and call $t_k = t_k^u = t_k^v$
4. if $u = t_k$ or $v = t_k$ then return $t_k$
5. if $t_{k+1}^u = u$ and not $isHead(u)$ then $c_u = u$
6. else $c_u = p(t_{k+1}^u)$
7. if $t_{k+1}^v = v$ and not $isHead(v)$ then $c_v = v$
8. else $c_v = p(t_{k+1}^v)$
9. if $l(c_u) < l(c_v)$ then return $c_u$
10. else return $c_v$
```

is ancestor of both u and v: line 3 can be easily implemented by scanning $list(u)$ and $list(v)$ together until the first different head is found. If neither u nor v coincides with the last common head (trivial case handled by line 4), we search $lca(u, v)$ in the relaxed solid path π with head t_k. Namely, the algorithm identifies two candidates c_u and c_v and returns the highest of them. Notice that node t_{k+1}^u is either the sentinel of $list(u)$ or the head following t_k in $list(u)$: in the former case (line 5 and Figure 8.1a) the candidate c_u is u itself, while in the latter case (line 6 and Figure 8.1b) the candidate is the parent of t_{k+1}^u. The candidate c_v is computed similarly and the algorithm returns the highest level node among c_u and c_v. We defer to Section 8.3.3 a formal proof of correctness.

We remark that all the information used by the algorithm are taken from $label(u)$ and $label(v)$. Then the query procedure is indeed a Decoder algorithm for an ILS.

8.3.3 Analysis

We will now prove that each update maintains a correct decomposition into relaxed solid paths in $O(\log^2 n)$ amortized running time and each query correctly computes the lowest common ancestor of any two nodes in $O(\log n)$ worst case running time.

Update. The insertion of a new node v as a child of a node u requires adding v to T, computing $\mathit{label}(v)$ and updating the decomposition into relaxed solid paths. Adding v to T requires constant time in order to update the children list of u. Computing $\mathit{label}(v)$ requires $O(\log n)$ worst case time since $\mathit{list}(u)$ must be copied and slightly modified. We recall that (from Property 8.5) $\mathit{list}(u)$ contains $O(\log n)$ integer values.

The most demanding step is to deal with changes in the path decomposition. The algorithm first scans $\mathit{list}(v)$ in order to identify all the heads t_i in the path from the root r down to v and to increment $|T_{t_i}|$ by one: since there are at most $2\lceil \log n \rceil$ such heads, this requires $O(\log n)$ time. Whenever a head t that no longer satisfies Inequality (8.2) is found, t is moved upward in the path decomposition and everything in T_t is recomputed: let us call this operation a *rebuild* of T_t.

Retrieving the head of π' and its rank, discriminating which case applies, and updating the path decomposition, require only constant time. The rebuild of T_t requires instead $O(|T_t| \log n)$ time: a traversal of T_t is performed recomputing the size of each subtree and the label of each node from the label of its parent (this requires $O(\log n)$ time by the upper bound on the label length). Let us now discuss how to amortize this cost on each update.

Let π be any solid path of rank j that has been created during some rebuild. We recall that the algorithm forces the head of π to satisfy Inequality 8.1 with respect to j. Hence, before a new rebuild takes place again at $\mathit{head}(\pi)$, at least $\alpha \cdot 2^{j+1}$ new nodes must have been added to its subtree. We amortize the cost of this new rebuild (i.e., $O(2^{j+1}(1+\alpha)\log n)$) over these insertions as follows: each insertion pays $O((1+1/\alpha)\log n) = O(\log n)$ credits for path π. Since, by Property 8.5, each node has $O(\log n)$ ancestral relaxed paths, its insertion will pay in total $O(\log^2 n)$ credits to amortize future rebuilds of all its ancestral paths. This proves that each update requires $O(\log^2 n)$ amortized running time.

Query. We first prove that the algorithm correctly finds the lowest common ancestor of any two nodes u and v. Apart from trivial cases handled by lines 1 and 2, in line 3 the algorithm identifies t_k, which is the lowest head

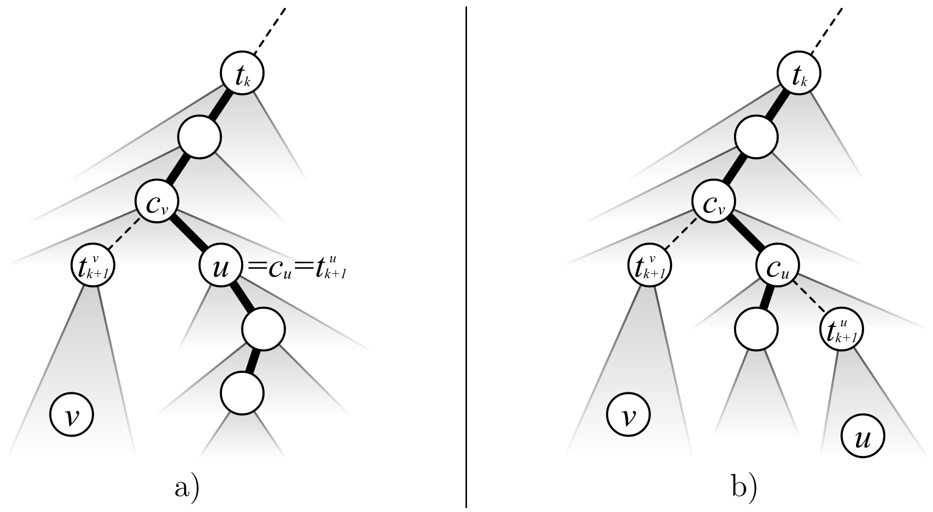


Figure 8.1: Two cases of $query(u, v)$: t_k is the lowest common head of u and v and c_v is the parent of t^v_{k+1}. a) $u \in \pi$: there are no heads following t_k in $list(u)$ (i.e., $t^u_{k+1} = u$ is the sentinel at the end of $list(u)$ and it is not a head). In this case $c_u = u$; b) $u \notin \pi$: in this case $c_u = p(t^u_{k+1})$.

of a relaxed solid path appearing both in $list(u)$ and in $list(v)$. It is not difficult to see that t_k is a common ancestor of u and v, but not necessarily the lowest one. Moreover, $lca(u, v)$ must belong to π, the relaxed solid path whose head is t_k: otherwise, there should have been another head, lower than t_k, appearing both in $list(u)$ and in $list(v)$.

If either u or v coincides with t_k, then it must be $t_k = lca(u, v)$: this case is considered in line 4. Otherwise, the algorithm identifies two candidates c_u and c_v, both in π, and returns the highest of them as the lowest common ancestor. The test in line 5 succeeds if and only if $u \in \pi$ (see Figure 8.1a): this case can be identified by checking if there are no other heads of relaxed solid paths between u and t_k ($t^u_{k+1} = u$, i.e., the scan has reached the sentinel triple) and if u itself is not a head (otherwise it would not be in π). In this case the candidate $c_u = u$. If $u \notin \pi$ (see Figure 8.1b), line 6 chooses the candidate c_u as the parent of t^u_{k+1} (the first head below t_k in $list(u)$). This candidate is the lowest node belonging to π in the ascending path from u to t_k. We compute c_v analogously.

Once we have identified the two candidates c_u and c_v, we compare their levels in order to select the highest one. This node is ancestor of both u and v. Let us assume without loss of generality that the selected candidate is c_v (the other case is symmetric). If $c_v = v$, there cannot be any common ancestor of u and v below v itself. If $c_v \neq v$, then the node below c_v in the path to v (i.e., t^v_{k+1}) is not an ancestor of u because it does not appear in $list(u)$. This proves that the selected candidate is the lowest common ancestor of u and v.

We now discuss the running time. Scanning the node lists (line 3) is the most expensive operation in the query algorithm described in Section 8.3.2. Since list lengths are bounded by $2\lceil \log n \rceil$ (Property 8.5), the query algorithm requires $O(\log n)$ time in the worst case.

8.4 A Concurrent Implementation

In this section we describe how to exploit the data structure presented in Section 8.3 in order to obtain a concurrent data structure for lowest common ancestor in dynamic trees.

Each label $label(v)$ has its own timestamp $timestamp(v)$ that represents last time the label has been computed or updated. Timestamps are especially useful when answering queries in order to check whether the information contained in the two node labels are reciprocally consistent.

Update. The update procedure used to add a new node v as a child of u is detailed in Program 2. Writing locks are exclusive, all other locks are shared to maximize concurrency. $children(u)$ represents the children list of node u.

Initially, the new node is added to its parent children list and its label is created starting from the label of its parent (lines 4 to 11). Subsequently, subtree sizes are incremented for all head in $list(v)$. Notice that t, ts_v, and ts_t are local (non shared) variables; also $list(v)$ used in line 12 is a local copy of the information contained in $label(v)$. If condition in line 16 is true, this means that a rebuild took place after v has been added. Then, everything in this subtree has been recomputed. Otherwise, the size update continues

Program 2 ConcurrentUpdate(v,u)

```
Acquire Exclusive Lock on $children(u)$
Add $v$ to $children(u)$
Release Exclusive Lock on $children(u)$

Create a label for $v$
Acquire Exclusive Lock on $label(v)$
Acquire Shared Lock on $label(u)$
Compute $label(v)$ from $label(u)$
$timestamp(v) = timestamp(u)$
Release Shared Lock on $label(u)$
Let $ts_v = timestamp(v)$
Release Exclusive Lock on $label(v)$

for each $t$ in $list(v)$ do
   Acquire Shared Lock on $label(t)$
   Let $ts_t = timestamp(t)$
   Release Shared Lock on $label(t)$
   if $ts_t > ts_v$ then return
   else
      Acquire Exclusive Lock on $|T_t|$
      $|T_t| = |T_t| + 1$
      Release Exclusive Lock on $|T_t|$
      if $t$ changes rank then
         $rebuild(T_t)$
         return
```

until a node whose rank changes (that no longer satisfy Inequality 8.2) is eventually encountered: here a rebuild takes place.

The function $rebuild(T_t)$ initially acquires an exclusive lock on $label(t)$, this will prevent any query to access information in T_t. Then, visiting the subtree, the decomposition into solid paths is recomputed, as well as the size of each subtree rooted in any head (proper locks are acquired on each part of the data structure being read or wrote). Another visit of T_t is performed to recompute all node labels. Timestamps are updated to reflect the rebuild, $timestamp(t)$ will be the instant in which labels recomputation started. Other node timestamps should satisfy the constraint that each node must have a timestamp greater or equal to the one of its parent. At the end the exclusive lock on *label*(t) is released. Each time the label of a node in T_t is written, an exclusive lock is acquired and then is released, to avoid undesired readings in the meanwhile.

Query. In order to answer a query (see Program 3) we make a local copy of u and v labels, we check their consistency by comparing their timestamps with those of t_i^u and t_{i+1}^u (and t_i^v and t_{i+1}^v respectively). If we find an ancestor whose timestamp is newer than the one of u (or v) this means that a rebuild took place and a new label for u (or v) has been computed (or is going to be computed). In this case the query fails; the process can try querying again (possibly after a little random delay). Notice that there is no need to check timestamps of nodes below t_{i+1}^u (or t_{i+1}^v) because, even if a rebuild took place in one of these subtrees, this would not affect the query. If timestamps are consistent the lca is computed as described in Section 8.3.2.

All locks are shared so that multiple queries can take place at the same time.

Program 3 ConcurrentQuery(u,v)

```
Acquire Shared Lock on $label(u)$ and $label(v)$
Make a local copy of $label(u)$ and $label(v)$
Release Shared Lock on $label(u)$ and $label(v)$

Scan $list(u)$ and $list(v)$ until $t_i^u = t_i^v$
Let $x$ be the last $t_i^u = t_i^v$
Acquire Shared Lock on $label(x)$
Let $ts_x = timestamp(x)$
Release Shared Lock on $label(x)$
if $ts_x > \min\{timestamp(u), timestamp(v)\}$ then fail

Let $y_u = t_{i+1}^u$ and $y_v = t_{i+1}^v$
Acquire Shared Lock on $label(y_u)$ and $label(y_v)$
Let $ts_{yu} = timestamp(y_u)$ and $ts_{yv} = timestamp(y_v)$
Release Shared Lock on $label(y_u)$ and $label(y_v)$
if $ts_{yu} > timestamp(u)$ or $ts_{yv} > timestamp(v)$ then fail

Compute $c_u$ and $c_v$ and return the one of minimum level
```

8.4.1 Analysis

We now discuss the correctness of our concurrent implementation. First of all, we remark that the data structure is deadlock free. Deadlocks can only be caused by exclusive locks, therefore the query procedure cannot produce any undesired situation. On the other hand, updates alway acquire locks in a top down order, so it is not possible that two updates produce a deadlock. If a rebuild on a certain node v is going on, then all updates that try to acquire a lock on v will be suspended (waiting for the lock to be acquired) until the end of the rebuild.

Two (or more) query operations can be executed concurrently without any problem and in any order. Indeed, queries only acquire shared locks and do not change the labels, therefore none of the two can affect the correctness of the other. If a query (on u and v) is executed concurrently with an update, some interaction may happen. More specifically, if the update causes a rebuild of a tree T_x containing the two nodes of the query, the labels of these nodes can be not reciprocally consistent. Indeed, the lists of heads of ancestral solid paths are changing and it may be the case that one label reflects this change while the other still does not. To prevent the query to produce a wrong result, timestamps are checked (see Program 3), and in case of inconsistency, the query fails. Two updates can be executed concurrently, indeed exclusive locks ensure that the operations are serialized. If two rebuilds (one on T_x and the other on T_y, with T_y subtree of T_x) are executed concurrently, they are serialized. Indeed, the rebuild on T_y holds an exclusive lock on y. When the rebuild of T_x will visit the tree, it will be not able to acquire a lock on y until the other rebuild is running: rebuild of T_x will be suspended, waiting for the lock on y to be released.

Let us now show the running time of the operations does not change. Here we also discuss how many locks each operation requires. A query acquires $O(\log n)$ shared locks. The operations required to compute the lca are exactly those used in Program 1: then the overall worst case running time is $O(\log n)$. Same reasoning applies for the update procedure: it requires $O(\log n)$. However, an update may imply a rebuild. As we have discussed in Section 8.4.1, the cost of each rebuild can be amortized resulting in an $O(\log^2 n)$ amortized running time of each update. The number of locks acquired by a rebuild operation, is linear in the size of the subtree being rebuilt. Therefore, the number of locks can be amortized on update operations with the very same argument used to the running time: each update requires $O(\log^2 n)$ locks (in an amortized sense).

8.5 Experimental Comparison

In this section we report details and results of an experimental analysis aimed to compare the labeling scheme proposed by Peleg and the one due to us. Peleg's labeling scheme is intended only for static trees and is not designed for a concurrent settings. For this reason we limited the analysis to static trees in a sequential setting: the features of our data structure have been only partially implemented in this investigation. In particular, we don't need to implement concurrency and all details related to our relaxed solid path decomposition (needed to handle dynamic trees).

Namely, in this section we compare three different solid path decompositions and the two label structures due to Peleg and to us. Different combinations of these two ingredients yield different labeling schemes: one of them coincides with the tree labeling scheme for least common ancestors originally proposed by Peleg. The main findings of our experiments can be summarized as follows.

- Among different path decompositions, those generating the smallest number of paths (with the largest average path length) appear to be preferable in order to minimize the total size of the data structure.

- A variant of Peleg's scheme proposed in [20] achieves the best performances in terms of space usage and construction time.

- Peleg's scheme, used with a minor variant of the path decomposition originally proposed in [87], exhibits the fastest query times.

- All the data structures are very fast in practice. Although node labels have size $O(\log^2 n)$ bits, only a small fraction of the labels is considered when answering random queries: typically, no more than a constant number of words per query is read in all our experiments. However, query times slightly increase with the instance size due to cache effects.

- Variants of the data structures carefully implemented with respect to alignment issues save 20% up to 40% of the space, but increase the query times approximately by a factor 1.3 on our data sets. The space saving reduces as the instance size gets larger.

8.5.1 Ingredients

As mentioned, we will compare three different solid path decompositions and two label structures. The first solid path decomposition is the one based on Large Child used by Peleg (described in Section 8.2). The second one is a minor variant of the first one, it is based on the concept of Maximum Child and is described below. The third one is the one based on Rank (as described in the first part of Section 8.3.1). There is no need to consider relaxed solid paths because we are not going to deal with dynamic trees. We remark that these decompositions have proven to be a useful tool for computing least common ancestors in different models [33, 34, 71, 87].

Decomposition by Maximum Child. This is a minor variant of the decomposition based on Large Child, that uses a relaxed definition of large nodes: a nonroot node v with parent u is considered a *maximum child* of u if $|T_v| = \max_{w:\ (u,w)\in T} |T_w|$. If two or more children of u satisfy this condition, ties are broken arbitrarily. The edge to the maximum child is considered as a solid edge. We note that a large node is necessarily a maximum child; however, a maximum child exists even when all the children v of a node u are such that $|T_v| \leq |T_u|/2$. All the basic properties of the decomposition by large child remain valid in this variant.

We combine these three decompositions with the two label structures due to Peleg (see Section 8.2) and to us (see Section 8.3.2) obtaining six different labeling schemes. Notice that, since each path decomposition ensures that each node has $O(\log n)$ ancestral solid path, all the obtained labeling schemes have maximum label size $\Theta(\log^2 n)$ bit.

From now on, we will refer to our label structure as CFP (from the name of the authors: Caminiti, Finocchi, and Petreschi).

8.5.2 Experimental Framework

In this section we describe our experimental framework, discussing implementation details of the data structures being compared, the performance indicators we consider, the problem instances generators, as well as our experimental setup. All implementations have been realized by the authors in ANSI C.

Data Structure Implementation Issues

As stated in Section 8.5.1, we implemented six labeling schemes. Each scheme comes in two variants, depending on alignment issues. In the `word` variant, every piece of information maintained in the node labels is stored at word-aligned addresses: some bytes are therefore used just for padding purposes. The actual sizes of node labels may be larger than the size predicted theoretically, but we expect computations on node labels to be fast. In the `bit` variant, everything is 1-bit aligned: this variant guarantees a very compact space usage, but requires operations for bit arithmetics that might have a negative impact on the running times of operations.

Performance Indicators

Main objectives we considered to evaluate the data structures include space usage, construction time, and query time. Space usage is strictly related to the length of the lists in the node labels, i.e., to the number of entries in such lists: besides the total size of the data structure (measured in MB, unless otherwise stated), we have therefore taken into account also the average and maximum list length. Other structural measures have been used to study the effect of the different path decompositions on the labeling schemes; among them, we considered the number of paths in which the tree is decomposed, the average and maximum length of paths, and the variance of path lengths.

Experimental Setup

Our experiments have been carried out on a workstation equipped with two Dual Core Opteron processors with 2.2 GHz clock rate, 6 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Debian (Kernel 2.6.8). All programs have been compiled through the GNU `gcc` compiler version 3.3.5 with optimization level `O3`, using the C99 revised standard of the C programming language.

Random values have been produced by the `rand()` pseudo-random source of numbers provided by the ANSI C standard library. We used only odd seeds to initialize the random generators; we randomly generated the sequence of seeds used in each test starting from a base odd seed.

In our experiments we averaged each data point on (at least) 1000 different instances. When computing running times of query operations, we averaged the time on (at least) 10^6 random queries. The running time of each experiment was measured by means of the standard system call `getrusage()` and, unless stated otherwise, is given in milliseconds.

8.5.3 Experimental Results

In this section we summarize our main experimental findings. We performed experiments using a wide variety of parameter settings and instance families, always observing the same relative performances of the data structures. For this reason we report the results of our experiments on trees generated uniformly at random. We remark that random trees are generated with the fast method based on tree encoding deeply described in Chapter 4 of this thesis.

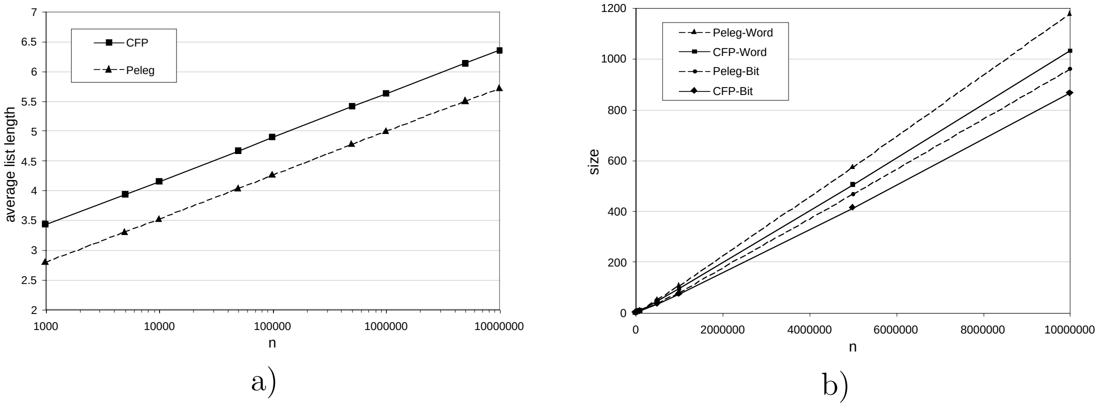
	maxChild	largeChild	rank
Number of paths	3678739	4172966	6803270
Average path length	2.72	2.4	1.47
Variance of path length	73.7	61.2	7.9
Maximum path length	15352	15351	6346
Average list length (Peleg)	5.72	5.89	12.40
Variance of list length (Peleg)	2.16	2.32	10.85
Maximum list length (Peleg)	15	15	24
Data structure size (Peleg)	1179	1203	2199
Average list length (CFP)	6.36	6.47	12.72
Variance of list length (CFP)	2.06	2.18	10.44
Maximum list length (CFP)	15	15	24
Data structure size (CFP)	1033	1045	1761

Table 8.1: Comparison of path decompositions. The results of this experiment are averaged over 250 random trees with $n = 10^7$ nodes. Only the word variant of the data structures is considered.

Path Decomposition

Our first aim was to analyze the effects of different path decomposition strategies on the size of node labels. A typical outcome of our experiments is exemplified in Table 8.1. With respect to all measures, maxChild (the decomposition by Maximum Child) appears to be slightly preferable than largeChild (the decomposition by Large Child) and considerably better than rank (the decomposition by Rank). Consider first the structural measures: among the three decompositions, maxChild generates the smallest number of solid paths. Paths are therefore longer on the average, and their lengths exhibit a higher variance. On the opposite side, the number of paths generated by rank is almost twice as large, and their length is almost twice as small.

The number of paths is strictly related to the number of solid heads in the lists associated to tree nodes, i.e., to the list lengths. Indeed, this is the case both for Peleg and CFP labels, not only when considering the maximum list length, but also on the average. In particular, Table 8.1 shows that the average list length appears to be inversely proportional to the path length: this can be easily explained by observing that if paths are longer on



a)

b)

Figure 8.2: Size comparison for `Peleg`'s and `CFP`'s schemes: a) average list length; b) total size, measured in MB.

the average, the number of paths in any root-to-leaf path is expected to be smaller, and so is the number of heads in the node labels.

We remark that for all the decompositions the average and the maximum list length for `Peleg` and `CFP` are very similar: the list length in the case of Peleg's scheme appears to be only slightly smaller, due to the presence of sentinel triples in `CFP`. Instead, the total size of the data structure is considerably smaller in the case of `CFP`: we will further discuss this aspect in the next section.

Size Comparison

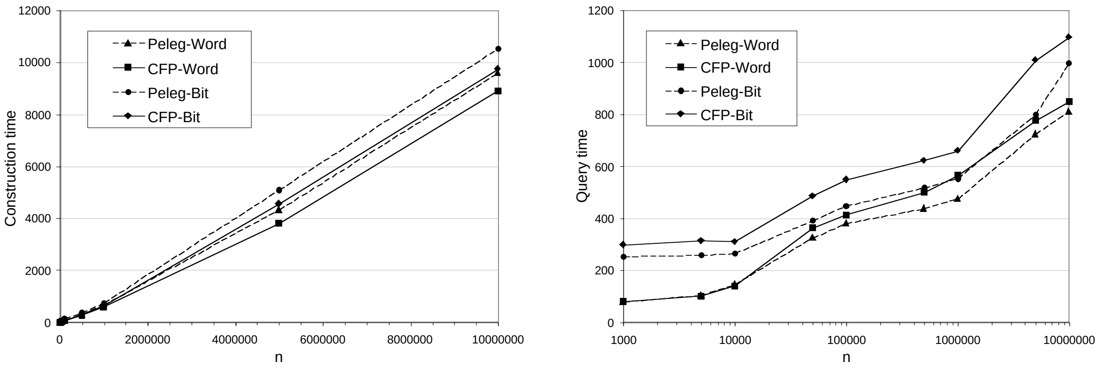
Our next aim is to evaluate the requirements of `Peleg`'s and `CFP`'s schemes with respect to the space usage. We will limit to use `maxChild` as a common path decomposition for both labeling schemes, since it proved to be consistently better than the other two decompositions in all the experiments we performed. To compare the size of the data structures, we measured both the average list length and the total data structure size on random trees with an increasing number of nodes. Figure 8.2 illustrates one such experiment, where the number n of nodes ranges from 10^3 to 10^7.

As already observed in Table 8.1, lists in `Peleg`'s scheme are shorter, but the total size of the data structure is larger than in `CFP`: this is because the lists are made of triples, instead of quadruples, and the smaller list length in Peleg's scheme is not sufficient to compensate for the presence of one more information in each element of the lists. We remark that lists are very short in practice for both schemes: they contain on the average 3 up to 6 elements for the data sets considered in this experiment. This value is very close to $\log_{10} n$, showing that the constant factors hidden by the asymptotic notation in the theoretical analysis are very small for the `maxChild` path decomposition. In Figure 8.2(b) we also distinguish between the `bit` and `word` versions of the data structures (there is no such difference with respect to the average list length): as expected, for both schemes the `bit` versions can considerably reduce the space usage. We will analyze further these data below.

Running Times

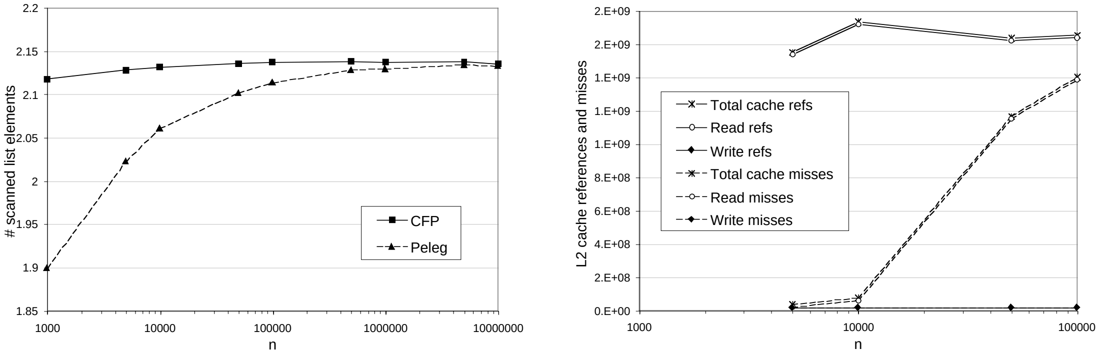
According to the theoretical analysis, the construction times and the query times for the different labeling schemes are asymptotically the same. A natural question is whether this is the case also in practice. Our experiments confirmed the theoretical prediction only in part, showing that the constant factors hidden by the asymptotic notation can be rather different for `Peleg`'s and `CFP`'s schemes. The charts in Figure 8.3, for instance, have been obtained on the same data sets used for the experiment reported in Figure 8.2: these charts show that `Peleg` is slower than `CFP` when considering initialization time, but faster when considering query times. The `bit` versions of the data structures are always slower than the corresponding `word` versions.

In order to explain the larger construction time of `Peleg`'s scheme, notice that `Peleg` makes use of a depth-first traversal of the tree, that is instead avoided by `CFP`: all the other operations performed by the initialization algorithms (i.e., path decomposition and list construction) are instead very similar. We also recall that Peleg's data structure is larger than `CFP`, and the size of a data structure is clearly a lower bound on its construction time. However, the larger amount of information maintained by `Peleg` in the node lists is efficiently exploited in order to get faster query times: as an example, if one of the two input nodes is ancestor of the other, the query algorithm



a) b)

Figure 8.3: Running time comparison for `Peleg`'s and `CFP`'s schemes: a) construction time; b) average query time.

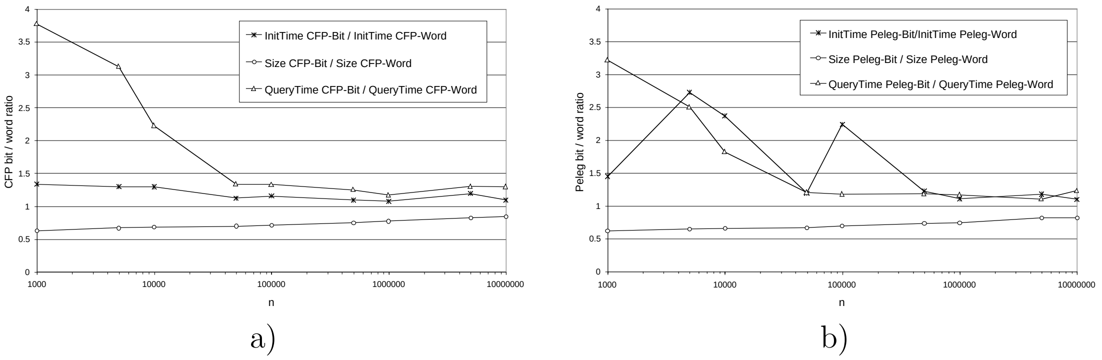


a) b)

Figure 8.4: a) Average number of list elements scanned by the query algorithms; b) number of references to L2 cache and number of cache misses.

used by `CFP` needs to scan the beginning of the node lists, while the depth-first intervals directly provide the answer in the case of `Peleg`'s data structure.

To get a deeper understanding of the query times, we also measured the average number of list elements scanned by the query algorithms during a sequence of operations. This number turns out to be very small both for `Peleg` and for `CFP`, as shown by the charts reported in Figure 8.4(a): on the average, slightly more than 2 elements are considered in each query even on the largest instances. `Peleg` considers less elements than `CFP`, especially for small values of n: on small trees, two nodes taken uniformly at random have indeed a higher probability to be one ancestor of the other, and for all



a) b)

Figure 8.5: Space saved by the `bit` versions and time saved by the `word` versions: a) `CFP`; b) `Peleg`.

these queries `Peleg` can avoid to scan the list at all, as we observed above. Quite surprisingly, however, for the largest values of n the number of scanned list elements remains almost constant for both data structures: this seems to be in contrast to the fact that the query times increase (see Figure 8.3), and suggests that the larger running times may be mainly due to cache effects. To investigate this issue, we conducted a preliminary experimental analysis of the number of cache misses incurred by the data structures using the `valgrind` profiler: the outcome of one such experiment is reported in Figure 8.4(b) and confirms that the number of L2 cache read misses increases with n, even if the number of cache references does not increase substantially.

Trading Space for Time

The experimental results discussed up to this point show that the `bit` versions of the data structures require more space than the corresponding `word` versions, but have larger construction and query times. In Figure 8.5 we summarize the space-time tradeoffs, both for `Peleg` and for `CFP`. The charts show the differences between `bit` and `word` versions tend to decrease for all measures as the instance size increases: this depends on the fact that, as n increases, the value $\log n$ becomes progressively closer to the word size specific of the architecture, and therefore the number of bits wasted by the `word` versions becomes smaller. The size of the `bit` versions ranges approximately from 60% up to 80% of the size of the `word` versions on our data sets. On the other side, construction and query times of the `bit` versions are approximately 1.3 times higher than the `word` versions for the largest values of n (for small values of n the ratio is even larger).

8.6 Concluding Remarks

In this chapter we have exploited the concept of Informative Labeling Scheme to devise a concurrent data structure that allow fast computation of the lowest common ancestor on dynamic trees. We want to explicitly remark how our data structure can be easily adapted to answer distance queries on trees. Given two nodes u and v, their levels (l_u and l_v) are explicitly maintained in the data structure. When we compute the lowest common ancestor w of u and v, we also explicitly obtain its level l_w. The distance between u and v is simply $d(u, v) = (l_u - l_w) + (l_v - l_w)$.

In future we plan to extend this data structure to handle dynamic trees that admit deletion of leaves and insertion of internal nodes (edge splitting). Moreover, the idea of using ILS to design concurrent data structure seems to be promising and deserves to be better investigated. Problems other than lowest common ancestor can be considered both on trees and other classes of graphs.

It would be finally interesting to run an experimental analysis of the labeling scheme proposed in this chapter on a real concurrent architecture.

Chapter 9

Conclusions and Future Work

In this thesis we studied algorithmic aspects related to bijective tree encodings and their generalizations. We proposed a unified approach that works for all Prüfer-like codes and a more generic scheme based on the transformation of a tree into a functional digraph suitable all bijective codes. By means of these ideas we devised optimal encoding and decoding sequential algorithms for many known codes as well as for interesting variants of known codes. We considered parallel algorithms for Prüfer-like codes on the EREW PRAM model. In all cases our results either match or improve the performances of the best previous known algorithms. We also described possible applications of these codes to Genetic Algorithms and to random tree generation. Specifically, in the field of Genetic Algorithms, our modified version of the Happy code has been experimentally proven to be the best known code for trees by Paulden and Smith [85].

We believe that, at the moment, this field does not offer further interesting algorithmic problems to investigate. More stimuli may eventually come from other applications requiring codes to satisfy new unstudied properties.

Shifting our attention from trees to their superclasses, we proposed a new bijective encode for k-trees. For the best of our knowledge, our work is the first one that presents linear time encoding and decoding algorithms. Our idea can be adapted to obtain bijective codes for both rooted and unrooted k-trees as well as for Rényi k-trees.

Looking at k-arch graphs (a superclass of k-trees), we discovered that the known result concerning the number of such graphs was wrong [72]. We corrected this result providing a recursive formula to compute the number of labeled k-arch graphs on n nodes. Unfortunately, this result does not help us in defining a bijective code for this class of graphs. This work deserves to be continued by investigating whether or not it is possible to obtain a closed formula for counting k-arch graphs. If this question can be settle affirmatively, an attempt to devise a bijective code can be made.

In the last chapter we focused on Informative Labeling Schemes: a way to associate to each node a rich label, so that it is possible to perform computations directly from labels. Since Informative Labeling Schemes naturally realize a localization of the information required to perform a computation, we decided to exploit them to design concurrent data structures. As a first example, we presented a concurrent data structure that allow fast computation of lowest common ancestors and distances on dynamic trees (leaves insertion only).

The results obtained in this initial investigation reveal that this is a promising idea. There are many possible directions for future work:

- we plan to extend our data structure to handle dynamic trees that admit deletion of leaves and insertion of internal nodes (edge splitting);
- this approach can be applied to other classes of graphs (e.g., k-trees, weighted trees, direct acyclic graphs, etc.);
- problems other than lowest common ancestor can be considered (e.g., routing, distance, flow, connectivity, etc.);

Finally, it would be interesting to run experimental analysis on a real concurrent architecture comparing classical concurrent data structures with those obtained by means of Informative Labeling Schemes.

Glossary

:: : with symbol "::" we denote the concatenation of two strings.

$[a, b]$: the interval of integers from a to b (a and b included).

$A \setminus B$: if B is a set this is the difference among sets: $A \setminus B = \{x \in A : x \notin B\}$; if $B = (b_1, b_2, \ldots, b_{|B|})$ is a string this is the difference among a set and a string $A \setminus B = \{x \in A : x \notin \{b_1, b_2, \ldots, b_{|B|}\}\}$.

$\binom{A}{k}$: be the set of all k-subset of the set A. $|\binom{A}{k}| = \binom{|A|}{k}$.

$adj(v)$: the set of nodes adjacent to v in a graph, i.e., $\{u : d(u, v) = 1\}$. If v has only one adjacent node u, $adj(v)$ may be used to refer to u itself rather than to the set $\{u\}$.

$d(u, v)$: the distance between nodes u and v, i.e., the number of edges in the shortest path between u and v. The distance of a node from itself is 0, i.e., $d(v, v) = 0$.

$deg(v)$: the degree of node v, i.e., $|adj(v)|$.

$l(v)$: bottom up level of node v, i.e., $l(v) = \max\{d(v, u) : u \text{ is a leaf in } T_v\}$. Each leaf has level 0.

T_v : the subtree of T rooted at node v.

$T \setminus v$: the tree obtained by removing a leaf v from a tree T.

k-Arch Graphs: a generalization of k-trees, see Chapter 7.

Cayely tree: a tree on n nodes labeled with distinct integers in $[1, n]$.

Center of a graph: a node minimizing the maximum distance from all other nodes. A tree has at least one and at most two centers.

k-**Clique**: a complete graph on k nodes.

Digraph: a directed graph $G = (V, E)$. Each edge $(u, v) \in E$ is oriented from u towards v.

EREW: Exclusive Read Exclusive Write.

Euler Tour: of a connected, directed graph G = (V, E) is a cycle that traverses each edge of graph G exactly once. The Euler Tour of a tree can be efficiently computed on a PRAM [59].

FIFO: First In First Out.

Functional Digraph: a graph $G = (V, E)$ is a functional digraph for a function g if and only if $g : V \to V$ and $E = \{(v, g(v)) : v \in V\}$.

Genetic Algorithm: a search heuristic that hinge upon the evolutionary ideas of natural selection and genetic (see Section 4.2).

LIFO: Last In First Out.

PRAM: Parallel Random Access Machine [59].

Rényi k-**Tree**: a k-tree on n nodes rooted in $\{n - k + 1, \ldots, n\}$, see Chapter 6.

Star: a graph $G = (V, E)$ having a single node c (the center) connected with all the others nodes and no other edges, i.e., $E = \{(c, v) : v \in V \setminus \{c\}\}$. Analogously, a tree of height 2. Analogously, a connected bipartite graph $G = (V', V'', E)$ with $|V'| = 1$.

String: a string over an alphabet Σ is a sequence $S = (s_1, s_2, \ldots, s_k)$ where each symbol $s_i \in \Sigma$. The length of the string is k and $S \in \Sigma^k$.

k-**Trees**: a generalization of trees, see Chapter 6.

Bibliography

[1] S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo, and T. Rauhe. Compact Labeling Scheme for Ancestor Queries. *SIAM Journal on Computing*, 35(6):1295–1309, 2006.

[2] S. Alstrup, P. Bille, and T. Rauhe. Labeling Schemes for Small Distances in Trees. *SIAM Journal on Discrete Mathematics*, 19(2):448–462, 2005.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest Common Ancestors: a Survey and a New Distributed Algorithm. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pages 258–264, 2002.

[4] S. Alstrup and M. Thorup. Optimal Pointer Algorithms for Finding Nearest Common Ancestors in Dynamic Trees. *Journal of Algorithms*, 35(2):169–188, 2000.

[5] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms. *SIAM Journal on Computing*, 18(3):499–532, 1989.

[6] L.W. Beineke and R.E. Pippert. On the Number of k-Dimensional Trees. *Journal of Combinatorial Theory*, 6:200–205, 1969.

[7] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest Common Ancestors in Trees and Directed Acyclic Graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[8] H.L. Bodlaender. A Tourist Guide Through Treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[9] H.L. Bodlaender. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[10] H.L. Bodlaender. A Partial k-Arboretum of Graphs with Bounded Treewidth. *Theoretical Computer Science*, 209:1–45, 1998.

[11] N. Bonichon, C. Gavoille, and A. Labourel. Short Labels by Traversal and Jumping. In *Proceedings of the 13th International Colloquium on Structural Information and Communication Complexity (SIROCCO'06)*, pages 143–156, 2006.

[12] V. Boppana, I. Hartanto, and W.K. Fuchs. Full Fault Dictionary Storage Based on Labeled Tree Encoding. In *Proceedings of the 14th IEEE VLSI Test Symposium*, pages 174–179, 1996.

[13] C.W. Borchardt. Uber eine der Interpolation Entsprechende Darstellung der Eliminations-Resultante. *Journal für die Reine und Angewandte Mathematik*, 57:111–121, 1860.

[14] M.A. Breuer. Coding the Vertexes of a Graph. *IEEE Transactions on Information Theory*, 12:148–153, 1966.

[15] M.A. Breuer and J. Folkman. An Unexpected Result on Coding the Vertices of a Graph. *Journal of Mathematical Analysis and Applications*, 20:583–600, 1967.

[16] S. Caminiti, N. Deo, and P. Micikevičius. Linear-time Algorithms for Encoding Trees as Sequences of Node Labels. To appear on *Congressus Numerantium*, 2007.

[17] 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)*, pages 339–348, 2004.

[18] S. Caminiti, I. Finocchi, and R. Petreschi. Concurrent Data Structures for Lowest Common Ancestors. *Manuscript*, July 2007.

[19] S. Caminiti, I. Finocchi, and R. Petreschi. Engineering Tree Labeling Schemes: a Case Study on Least Common Ancestors. *Manuscript*, November 2007.

[20] S. Caminiti, I. Finocchi, and R. Petreschi. On Coding Labeled Trees. *Theoretical Computer Science*, 382(2):97–108, 2007.

[21] S. Caminiti and E.G. Fusco. On the Number of Labeled k-Arch Graphs. *Journal of Integer Sequences*, 10(7), 2007.

[22] S. Caminiti, E.G. Fusco, and R. Petreschi. Bijective Linear Time Coding and Decoding for k-Trees. *Submitted to Theory of Computing Systems.*

[23] S. Caminiti, E.G. Fusco, and R. Petreschi. A Bijective Code for k-Trees with Linear Time Encoding and Decoding. In *Proceedings of the Proceedings of the IntErnational Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, 2007.

[24] S. Caminiti and R. Petreschi. A General Scheme for Coding and Decoding Trees with Locality and Heritability. *Submitted to SIAM Journal of Discrete Mathematics.*

[25] 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)*, pages 251–262, 2005.

[26] A. Cayley. A Theorem on Trees. *Quarterly Journal of Mathematics*, 23:376–378, 1889.

[27] H.C. Chen and Y.L. Wang. An Efficient Algorithm for Generating Prüfer Codes from Labelled Trees. *Theory of Computing Systems*, 33:97–105, 2000.

[28] W.Y.C. Chen. A Coding Algorithm for Rényi Trees. *Journal of Combinatorial Theory*, 63A:11–25, 1993.

[29] W.Y.C. Chen. A General Bijective Algorithm for Increasing Trees. *Systems Science and Mathematical Sciences*, 12:194–203, 1999.

[30] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 937–946, 2002.

[31] H. Cohen. PARI/GP Development Headquarter. Website: http://pari.math.u-bordeaux.fr/.

[32] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[33] R. Cole and R. Hariharan. Dynamic LCA Queries on Trees. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 235–244, 1999.

[34] R. Cole and R. Hariharan. Dynamic LCA Queries on Trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[35] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[36] N. Deo, N. Kumar, and V. Kumar. Parallel Generation of Random Trees and Connected Graphs. *Congressus Numerantium*, 130:7–18, 1998.

[37] N. Deo and P. Micikevičius. Parallel Algorithms for Computing Prüfer-Like Codes of Labeled Trees. Technical report, CS-TR-01-06, Department of Computer Science, University of Central Florida, Orlando, 2001.

[38] N. Deo and P. Micikevičius. Prüfer-like codes for labeled trees. *Congressus Numerantium*, 151:65–73, 2001.

[39] 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.

[40] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York, 1986.

[41] 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.

[42] W. Edelson and M.L. Gargano. Modified Prüfer code: $O(n)$ implementation. *Graph Theory Notes of New York*, 40:37–39, 2001.

[43] 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.

[44] Ö. Eğecioğlu and J.B. Remmel. A Bijection for Spanning Trees of Complete Multipartite Graphs. *Congressus Numerantium*, 100:225–243, 1994.

[45] Ö. Eğecioğlu, J.B. Remmel, and G. Williamson. A Class of Graphs which has Efficient Ranking and Unranking Algorithms for Spanning Trees and Forests. *International Journal of Foundations of Computer Science*, 15(4):619–648, 2004.

[46] Ö. Eğecioğlu and L.P. Shen. A Bijective Proof for the Number of Labeled q-Trees. *Ars Combinatoria*, 25B:3–30, 1988.

[47] D. Foata. Enumerating k-Trees. *Discrete Mathematics*, 1(2):181–186, 1971.

[48] V.K. Garg and A. Agarwal. Distributed Maintenance of a Spanning Tree Using Labeled Tree Encoding. In *Proceedings of 11th International Euro-Par Conference*, pages 606–616, 2005.

[49] C. Gavoille, M. Katz, N.A. Katz, C. Paul, and D. Peleg. Approximate Distance Labeling Schemes. In *Proceedings of the 9th European Symposium on Algorithms (ESA'01)*, pages 476–487, 2001.

[50] C. Gavoille, D. Peleg, S. Perennes, and R. Raz. Distance Labeling in Graphs. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 210–219, 2001.

[51] J. Gottlieb, B.A. Julstrom, G.R. Raidl, and F. Rothlauf. Prüfer Numbers: A Poor Representation of Spanning Trees for Evolutionary Search. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 343–350, San Francisco, California, USA, 2001.

[52] C. Greene and G.A. Iba. Cayley's Formula for Multidimensional Trees. *Discrete Mathematics*, 13:1–11, 1975.

[53] R. Greenlaw, M.M. Halldórsson, and R. Petreschi. On Computing Prüfer Codes and Their Corresponding Trees Optimally in Parallel. In *Proceedings of Journes de l'Informatique Messine (JIM'00)*, Metz, France, 2000.

[54] R. Greenlaw and R. Petreschi. Computing Prüfer Codes Efficiently in Parallel. *Discrete Applied Mathematics*, 102(3):205–222, 2000.

[55] Y. Han and X. Shen. Parallel Integer Sorting is More Efficient than Parallel Comparison Sorting on Exclusive Write PRAMS. *SIAM Journal on Computing*, 31(6):1852–1878, 2002.

[56] F. Harary and J.P. Hayes. Edge Fault Tolerance in Graphs. *Networks*, 23:135–142, 1993.

[57] F. Harary and E.M. Palmer. On Acyclic Simplicial Complexes. *Mathematika*, 15:115–122, 1968.

[58] D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[59] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[60] A. Joyal. Une Theorie Combinatoire des Series Formelles. *Advances in Mathematics*, 42:1–81, 1981.

[61] B.A. Julstrom. The Blob Code: A Better String Coding of Spanning Trees for Evolutionary Search. In *Representations and Operators for Network Problems (ROPNET 2001)*, pages 256–261, San Francisco, California, USA, 2001.

[62] B.A. Julstrom. The Blob Code is Competitive with Edge-Sets in Genetic Algorithms for the Minimum Routing Cost Spanning Tree Problem. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO'05)*, pages 585–590, Washington DC, USA, 2005.

[63] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. In *Proceedings of the 20th ACM Symposium on Theory of Computing (STOC'88)*, pages 334–343, 1988.

[64] H. Kaplan and T. Milo. Short and Simple Labels for Small Distances and other Functions. In *Proceedings of the 7th Workshop on Algorithms and Data Structures (WADS'01)*, pages 246–257, 2001.

[65] H. Kaplan, T. Milo, and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 954–963, 2002.

[66] M. Katz, N.A. Katz, A. Korman, and D. Peleg. Labeling Schemes for Flow and Connectivity. *SIAM Journal on Computing*, 34(1):23–40, 2004.

[67] M. Katz, N.A. Katz, and D. Peleg. Distance Labeling Schemes for Well-Separated Graph Classes. In *Proceedings of the 17th Symposium on Theoretical Aspects of Computer Science (STACS'00)*, pages 516–528, 2000.

[68] P. Klingsberg. *Doctoral Dissertation.* PhD thesis, University of Washington, Seattle, Washington, 1977.

[69] D.E. Knuth. Oriented Subtrees of an Arc Digraph. *Journal of Combinatorial Theory*, 3:309–314, 1967.

[70] D.E. Knuth. *The Generating All Trees – History of Combinatorial Generation*, volume 4(4) of *Art of Computer Programming.* Addison-Wesley, 2006.

[71] T. Kopelowitz and M. Lewenstein. Dynamic Weighted Ancestors. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA'07)*, pages 565–574, 2007.

[72] C. Lamathe. The Number of Labelled k-Arch Graphs. *Journal of Integer Sequences*, 7, 2004.

[73] 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.

[74] P. Micikevičius. *Parallel Graph Algorithms for Molecular Conformation and Tree Codes.* PhD thesis, University of Central Florida, 2002.

[75] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[76] M. Moir and N. Shavit. Concurrent Data Structures. In *Handbook of Data Structures and Applications.* CRC Press, 2005.

[77] J.W. Moon. The Number of Labeled k-Trees. *Journal of Combinatorial Theory*, 6:196–199, 1969.

[78] J.W. Moon. *Counting Labeled Trees.* William Clowes and Sons, London, 1970.

[79] E.H. Neville. The Codifying of Tree-Structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.

[80] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms.* Academic Press, New York, 1978.

[81] J.B. Orlin. Line-Digraphs, Arborescences, and Theorems of Tutte and Knuth. *Journal of Combinatorial Theory*, 25:187–198, 1978.

[82] C.C. Palmer and A. Kershenbaum. Representing Trees in Genetic Algorithms. In *First IEEE Conference on Evolutionary Computation*, pages 379–384, Orlando, FL, 1994.

[83] T. Paulden and D.K. Smith. The Rainbow Code: A Superior Genetic Algorithm Representation for Layered Trees. In *Proceedings of the 34th International Conference on Computers and Industrial Engineering*, 2004.

[84] T. Paulden and D.K. Smith. From the Dandelion Code to the Rainbow Code: A Class of Bijective Spanning Tree Representations With Linear Complexity and Bounded Locality. *IEEE Transactions on Evolutionary Computation*, 10(2):108–123, 2006.

[85] T. Paulden and D.K. Smith. Recent Advances in the Study of the Dandelion Code, Happy Code, and Blob Code Spanning Tree Representations. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2111–2118, 2006.

[86] D. Peleg. Proximity-Preserving Labeling Schemes and Their Applications. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'99)*, pages 30–41, 1999.

[87] D. Peleg. Informative Labeling Schemes for Graphs. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00)*, pages 579–588, 2000.

[88] D. Peleg. Informative Labeling Schemes for Graphs. *Theoretical Computer Science*, 340(3):577–593, 2005.

[89] S. Picciotto. *How to Encode a Tree*. PhD thesis, University of California, San Diego, 1999.

[90] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:142–144, 1918.

[91] Raidl and B.A. Julstrom. Edge-sets: An Effective Evolutionary Coding of Spanning Trees. *IEEE Transactions on Evolutionary Computation*, 7(3):225–239, 2003.

[92] C.R. Reeves and J.E. Rowe. *Genetic Algorithms: A Guide to GA Theory*. Springer, 2003.

[93] J.B. Remmel and G. Williamson. Spanning Trees and Function Classes. *Electronic Journal of Combinatorics*, R24, 2002.

[94] 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.

[95] D.J. Rose. On Simple Characterizations of k-Trees. *Discrete Mathematics*, 7:317–322, 1974.

[96] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, Heidelberg, Germany, 2002.

[97] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[98] N.J.A. Sloane and S. Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, 1995. http://www.research.att.com/∼njas/sequences.

[99] M. Thorup. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 242–251, 2001.

[100] P. Todd. A k-Tree Generalization that Characterizes Consistency of Dimensioned Engineering Drawings. *SIAM Journal Discrete Mathematics*, 2:255–261, 1989.

[101] E. Tompson, T. Paulden, and D.K. Smith. The Dandelion Code: A New Coding of Spanning Trees for Genetic Algorithms. *IEEE Transactions on Evolutionary Computation*, 11(1):91–100, 2007.

[102] W. Tutte. The Dissection of Equilateral Triangles into Equilateral Triangles. In *Proceedings of Cambridge Philosophical Society*, volume 44, pages 463–482, 1948.

[103] I. Vardi. *Computational Recreations in Mathematica*, chapter Computing Binomial Coefficients. Addison-Wesley, Redwood City, CA, 1991.

[104] Y.L. Wang, H.C. Chen, and W.K. Liu. A Parallel Algorithm for Constructing a Labeled Tree. *Transactions on Parallel and Distributed Systems*, 8(12):1236–1240, 1997.

[105] Xingzhi Wen and Uzi Vishkin. PRAM-on-Chip: First Commitment to Silicon. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pages 301–302, 2007.

[106] G. Zhou and M. Gen. A Note on Genetic Algorithms for Degree-Constrained Spanning Tree Problems. *Networks*, 30:91–95, 1997.