J. Parallel Distrib. Comput. 70 (2010) 1119–1127

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Unified parallel encoding and decoding algorithms for Dandelion-like codes[☆]

Saverio Caminiti [*], Rossella Petreschi

Computer Science Department, Sapienza University of Rome, Via Salaria, 113 - I00198 Rome, Italy

ARTICLE INFO

Article history:
Received 12 June 2009
Received in revised form
9 July 2010
Accepted 10 July 2010
Available online 17 July 2010

Keywords:
Bijective tree encoding
Prüfer code
Dandelion-like codes
PRAM algorithms

ABSTRACT

The Dandelion-like codes are eight bijections between labeled trees and strings of node labels. The literature contains optimal sequential algorithms for these bijections, but no parallel algorithms have been reported. In this paper the first parallel encoding and decoding algorithms for Dandelion-like codes are presented. Namely, a unique encoding algorithm and a unique decoding algorithm, which when properly parameterized, can be used for all Dandelion-like codes, are designed. These algorithms are optimal in the sequential setting. The encoding algorithm implementation on an EREW PRAM is optimal, while the efficient implementation of the decoding algorithm requires concurrent reading.

1. Introduction

Trees are one of 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, and 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). In the literature a range of different representations of tree data structures can be found.

This paper focuses on those representations based on coding labeled trees by means of strings of node labels. It is well known that a naïve method for relating a tree to a string is to associate each node with its parent. Unfortunately, however, the resulting code is not bijective, as an arbitrary string can represent a cyclic and non-connected graph.

At the beginning of the last century, Prüfer [32] showed that it is possible to obtain a one-to-one association between trees and strings, thus providing the first bijective code. Since then many researchers have been fascinated by this topic and have presented a variety of bijective codes. Only in the last two decades, these codes have been used in practical applications.

Bijective codes make it possible to generate random uniformly distributed trees and random connected graphs [13]: the generation of a random string followed by the use of a fast decoding algorithm is typically more efficient than generating a tree by adding edges randomly, since in the latter case one must be careful not to introduce cycles. Furthermore, constraints on the number and on the set of leaves, on the choice of the root, and on the degree of nodes can be easily imposed during the generation.

In XML databases, documents are nested structures often modeled as labeled trees. It is possible to transform XML data into sequences of labels using bijective tree codes. During the query process, search patterns are encoded as sequences as well, thus reducing the problem to substring matching [33]: this allows the total amount of data that needs to be searched during the query process to be reduced.

Genetic algorithms solving problems on trees maintain a population of data structures that represents candidate solutions for a given problem [34]. The association between structures and solutions is realized through a decoder which must exhibit certain desirable properties (like efficiency, locality, and heritability) in order for the evolutionary search to be effective [18].

In network tomography, starting from a matrix of Origin–Destination pair measurements, a researcher may wish to infer the logical network topology. When the network is a tree, it is possible to create a correspondence between the matrix and a string-based representation of the logical network. This leads to a provably correct algorithm for this problem [36].

Other fields of application in which bijective tree codes are used include: fault dictionary storage [2], distributed spanning tree maintenance [20], cryptographic secret sharing [30], and ant-colony optimization heuristics [1].

[☆] A preliminary version of the results presented in this paper appeared in [S. Caminiti, R. Petreschi, Parallel algorithms for Dandelion-like codes, in: Proceedings of the 9th International Conference on Computational Science, ICCS'09, in: LNCS, vol. 5544, 2009, pp. 611–620] [9].

[*] Corresponding author.
E-mail addresses: caminiti@di.uniroma1.it (S. Caminiti), petreschi@di.uniroma1.it (R. Petreschi).

Table 1
Costs of known algorithms for bijective string-based codes. Parallel algorithms are all designed for the EREW PRAM model, except for the Blob decoding requiring the CREW model. Costs are expressed as the number of processors multiplied by the maximum time required by a single processor.

		Sequential		Parallel	
		Encoding	Decoding	Encoding	Decoding
Prüfer-like	Prüfer [32]	$O(n)$ [24]	$O(n)$ [17]	$O(n)$ [21]	$O(n \log n)$ [5,38]
	2nd Neville [28]	$O(n)$ [26]	$O(n)$ [5]	$O(n\sqrt{\log n})$ [5]	$O(n\sqrt{\log n})$ [5]
	3rd Neville [28]	$O(n)$ [28]	$O(n)$ [28]	$O(n)$ [5]	$O(n\sqrt{\log n})$ [5]
	Stack-Queue [14]	$O(n)$ [14]	$O(n)$ [14]	$O(n\sqrt{\log n})$ [5]	$O(n\sqrt{\log n})$ [5]
	Chen [11]	$O(n)$ [6]	$O(n)$ [6]	$O(n)$ [6]	$O(n \log n)$ [6]
	Blob [31]	$O(n)$ [7]	$O(n)$ [7]	$O(n)$ [8]	$O(n \log n)$ [8]
	Happy [31]	$O(n)$ [31]	$O(n)$ [31]		
	Dandelion [31]	$O(n)$ [31]	$O(n)$ [31]		
	ϑ_n bijection [19]	$O(n)$ [31]	$O(n)$ [31]		
	MHappy [7]	$O(n)$ [7]	$O(n)$ [7]		

1.1. String of node labels

Unless stated otherwise, trees are rooted at node 0 and their n nodes are uniquely labeled from 0 to $n-1$.

Cayley's theorem establishes that the number of labeled trees of n nodes is n^{n-2} [10], so we consider bijections between such trees and strings over $[0, n-1]$ of length $n-2$.

In his proof of Cayley's theorem, Prüfer presented the first bijective string-based code for trees [32]. Many codes behaving like the Prüfer one (i.e., based on recursive leaf elimination) have been introduced: the Second and Third Neville codes [28] (the First Neville code is analogous to the Prüfer code); Moon's variations of Prüfer's and Neville's codes for rooted trees [27]; the Stack-Queue code due to Deo and Micikevičius [14]. These codes are usually called Prüfer-like codes. We have recently shown [6] that the Chen code [11] also falls into this class.

Other bijective codes with behaviors different from the Prüfer one have also been introduced: the ϑ_n bijection by Eğecioğlu and Remmel [19]; the code due to Kreweras and Moszkowski [25]; the Blob code, the Happy code, and the Dandelion code due to Picciotto [31]; the MHappy code due to Caminiti and Petreschi [7].

1.2. Algorithmic results

Even thought the Prüfer code is dated 1918, to the best of our knowledge, the first linear-time algorithm for encoding a tree appeared only in the late 70's and is due to Klingsberg [24]. The same author also designed an optimal decoding algorithm [17].

A straightforward implementation of the Second Neville code [28] would involve several calls to a sorting routine: this requires $O(n^2)$ running time using integer sort. Deo and Micikevičius [15] reduced the running time to $O(n \log n)$ with an encoding algorithm that uses a set of sorted lists. Micikevičius presented the first linear-time algorithm for the Second Neville code [26] while the first optimal decoding algorithm is due to Caminiti et al. [5]. A straightforward implementation of the Third Neville code [28] leads to linear-time algorithms. Deo and Micikevičius introduced the Stack-Queue code [14] with the explicit intent of providing a bijective code with simple linear-time algorithms. Unified optimal encoding and decoding algorithms for these Prüfer-like codes are given by Caminiti et al. [5]. Caminiti and Petreschi presented linear-time algorithms for the Chen code [6].

Among the three codes introduced by Picciotto optimal algorithms were proposed in [31] for just two of them. The first version of optimal algorithms for the Blob code was shown in [7], together with new optimal algorithms for the Happy code and the Dandelion code and the new MHappy code. The ϑ_n bijection is analogous to the Dandelion code and the reinterpretation of the Blob code given in [7] leads to the conclusion that it is identical to the Kreweras and Moszkowski code (see [29]).

A survey on optimal algorithms for bijective codes can be found in [4] while for a deeper dissertation we refer to [3].

Greenlaw, Halldórsson, and Petreschi [21] presented an optimal EREW PRAM algorithm for Prüfer encoding. It improves an earlier algorithm by Greenlaw and Petreschi [22] and, slightly changed, also works for the third Neville code [5] and for the Chen code [6]. Efficient, but non-optimal, parallel encoding algorithms for the Second Neville code and the Stack-Queue code were presented in [16] and improved in [5]. For the decoding phase, only non-optimal algorithms are known [5]. Recently, parallel algorithms for Blob code have been presented [8]: the encoding algorithm is optimal, while the decoding algorithm requires $O(n \log n)$ cost (number of processors multiplied by the maximum time required by a single processor) and concurrent read.

To the best of our knowledge no parallel algorithms for other bijective codes have been investigated. Table 1 summarizes the costs of sequential and parallel algorithms for encoding and decoding bijective codes known previously.

This paper is organized as follows: after a few preliminary definitions (Section 2), Section 3 examines a General Scheme defining bijections between trees and strings. Section 4 describes the Dandelion code as originally introduced and reinterprets this code as a transformation of the tree into a functional digraph to map it into the General Scheme. The rest of the paper introduces encoding algorithm and decoding algorithms that, properly parameterized, can be used for all Dandelion-like codes. In Section 5 the class of Dandelion-like codes is defined and the encoding and decoding algorithms are designed. In Section 6 the parallelization of these algorithms on a PRAM is detailed. Conclusions and open problems will follow.

2. Preliminary definitions

In this section, some definitions that will be used in the rest of the paper are reported.

Definition 1. Given a function g from the set $[0, n]$ to the set $[0, n]$, the *functional digraph* $G = (V, E)$ associated with g is a directed graph with $V = \{0, \dots, n\}$ and $E = \{(v, g(v))$ for every $v \in V\}$.

For this class of graphs the following proposition holds:

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

Corollary 1. *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 Fig. 1 (a)).*

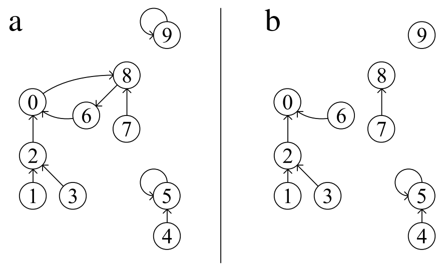

Fig. 1. (a) A functional digraph associated with a fully defined function; (b) A functional digraph associated with a function undefined in 0, 8, and 9.

Functional digraphs are easily generalizable for representing functions which are undefined in some values: if $g(x)$ is not defined, node x in G does not have outgoing edges. The connected component of G containing a node x such that $g(x)$ is not defined is a tree rooted at x without cycles or loops (see Fig. 1b). In the following the notation $u \rightsquigarrow v$ is used to identify the unique simple path from node u to node v.

Definition 2. A *labeled n-tree* is an unrooted tree with n nodes labeled with distinct values selected in the set $[0, n-1]$.

Remark 1. Let T be a rooted labeled tree and $p(v)$ be the parent of v for each v in T. T is the functional digraph associated with the function p.

Definition 3. A code associate trees with strings in such a way that different trees yield different strings. A *bijective code* is a code associating n-trees to $(n-2)$-strings, where an *n-string* is a string of n elements over the alphabet $[0, n+1]$.

A naïve method for relating a labeled tree (rooted at node 0) to a string C associates each node x with its parent $p(x)$: C is a string over the alphabet $[0, n-1]$ whose i-th element is $p(i)$. C has cardinality $n-1$ since node 0 (the root) has no parent and can be omitted. This code is not bijective. Indeed, an arbitrary string of length $n-1$ over the alphabet $[0, n-1]$ can represent a cyclic and non-connected graph and consequently does not necessarily correspond to a labeled tree.

This paper deals with bijective codes for labeled trees (simply trees). From now on, all tree edges will be considered as oriented upward from a node to its parent.

Below, when no confusion arises, we will identify a tree with its associated string, and vice versa. We will also consider vectors as functions and vice versa.

3. General scheme

Caminiti and Petreschi [7] defined a General Scheme for bijections between the set of labeled n-trees and the set of $(n-2)$-strings. Here this General Scheme is briefly recalled underlining that the main idea is to modify the naïve method to reduce the length of the string that it yields.

In order to build an $(n-2)$-string, the tree is rooted at a fixed node x, and its topology is manipulated in such a way that there exists another fixed node y having x as parent. When these conditions hold, the information related to both x and y can be omitted from the parent vector, thus obtaining an $(n-2)$-string.

While it is easy to root a given unrooted tree at a fixed node x, it is not clear how to guarantee the existence of edge (y, x).

To this purpose, a function φ that manipulates the tree in order to ensure the existence of (y, x) is required. This function must transform T into a functional digraph G associated with a function g. The value of g in x must be undefined and $g(y)$ must be equal to x. Given φ, x, and y the encoding scheme is as follows:

Algorithm: General Encoding Scheme
Input: an n-tree T
Output: an $(n-2)$-string C

```
1. root $T$ in $x$.
2. construct $G = \varphi(T)$.
3. for $v = 0$ to $n-1$ do
4.   if $v \neq x$ and $v \neq y$ then
5.     add $g(v)$ at the end of $C$.
```

For the coding to be bijective, the function φ must be invertible. Indeed, only under this hypothesis, it is possible to define the decoding scheme:

Algorithm: General Decoding Scheme
Input: an $(n-2)$-string C
Output: an n-tree T

```
1. for $v = 0$ to $n-1$ do
2.   if $v \neq x$ and $v \neq y$ then
3.     extract the first element $u$ from $C$;
4.     $g(v) = u$.
5. reconstruct the functional digraph $G$ associated to $g$.
6. add edge $(y, x)$ to $G$.
7. compute $T = \varphi^{-1}(G)$.
```

For each node from 0 to $n-1$ this scheme considers the node's outgoing edge, so the topology of graph G directly identifies the codeword C that represents G. The computational complexities of encoding and decoding schemes strictly depend on the computations of φ and φ^{-1}.

The idea of transforming a tree into a functional digraph was also used in [19,31]. The General Scheme provide a uniform framework that we exploit to design sequential and parallel algorithms for the class of Dandelion-like codes (see Section 5). This class contains the Dandelion code and other known codes, namely, the Happy code, the MHappy code, and the ϑ_n bijection. Since Dandelion-like codes are defined by modifying the behavior of the Dandelion code, next section is devoted to a detailed description of this code.

4. The Dandelion code

This section examines the Dandelion code as introduced by Picciotto, and reinterprets this code as a transformation of the tree into a functional digraph, in order to map it into the General Scheme.

The poetic name Dandelion derives from the fact that during the encoding all the nodes are connected to node 1 and the resulting tree looks like a dandelion flower (see example in Fig. 2(b)). During this process, labels on edges are introduced; these labels will be used to create the code. The original encoding algorithm for the Dandelion code is as follows:

Algorithm: Dandelion Encoding Algorithm
Input: an n-tree T rooted in 0 represented by the parent vector p
Output: an $(n-2)$-string C

```
1. root $T$ in 0.
2. for $v = n$ downto 2 do
3.   $h = p(v)$;
4.   $k = p(1)$;
5.   delete $(v, h)$;
6.   add $(v, 1)$ with label $label(v, 1) = h$.
7.   if a cycle has been created then
8.     delete $(1, k)$;
9.     add $(1, h)$;
10.    $label(v, 1) = k$.
11. for $v = 2$ to $n$ do
12.   $C(v-2) = label(v, 1)$.
```

Analyzing the algorithm, we see that the complexity of the Dandelion Encoding Algorithm is not linear, because it requires testing whether a cycle has been introduced at each iteration. Moreover, for each node v the corresponding value in C is $p(v)$ unless v satisfies the test in Line 7. For this reason we will focus on those nodes that satisfy this test and call them *flying* nodes.

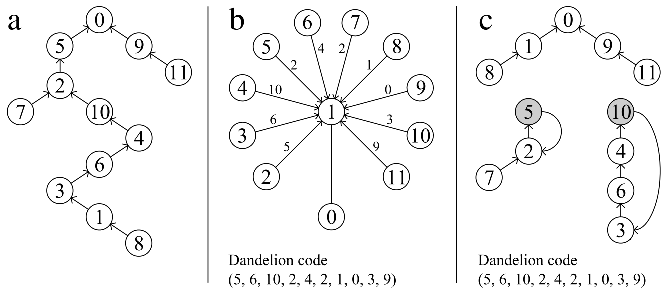

Fig. 2. (a) A sample tree T rooted in 0; (b) T after the execution of the Dandelion Encoding Algorithm. (c) The digraph $G = \varphi_d(T)$, flying nodes are depicted in grey.

Lemma 1. *Flying nodes are all nodes v such that:*

1. $v \in 1 \rightsquigarrow 0$ *(excluding* 1 *and* 0 *) and*
2. $v = max\{w \in v \rightsquigarrow 0\}$.

Proof. Condition 1 trivially holds, otherwise test of Line 7 cannot be true.

Given $v \in 1 \rightsquigarrow 0$ (except 1 and 0), let $\mu = max\{w \in v \rightsquigarrow 0\}$. Obviously, μ is not smaller than v since v belongs to $v \rightsquigarrow 0$. Assume that v is a flying node but $\mu > v$. μ is processed before v since nodes are considered by the algorithm in decreasing order. Node μ is connected to 1, therefore it introduces a cycle containing v. When the cycle is broken (Lines 8 and 9), all the nodes in the cycle are excluded by the resulting path $1 \rightsquigarrow 0$. This implies that in the following steps v cannot be a flying node: a contradiction. So if v is a flying node then $\mu = v$.

On the other hand, if $v = \mu$, v will be in $1 \rightsquigarrow 0$ when it is processed by the algorithm and thus v introduces a cycle. □

When a cycle is broken (Line 8), node 1 is connected to h, the former parent of the flying node v inducing the cycle, and the label of edge $(v, 1)$ becomes k (the former parent of 1). In string C, the element corresponding to v will contain the value k. In order to map the Dandelion code into the General Scheme, it is possible to avoid edge labels and to generate C directly from vector p assigning $p(v) = k$ for each flying nodes v.

Let us define a function φ_d which, given a tree T, constructs a graph G. φ_d considers flying nodes of T in decreasing order and swaps $p(v)$ and $p(1)$ for each flying node v. We defer an example of the transformation induced by φ_d to the end of this section.

Theorem 1. *The 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 $g(1) = 0$. Lemma 1 guarantees that the code generated using φ_d is the same code generated using Dandelion Encoding Algorithm.

Now it is to prove that φ_d is invertible, i.e., it is to show how to rebuild T from G. Flying nodes must be recomputed and all cycles (and loops) in G must be broken.

Each cycle Γ is identified together with its maximum node γ. Clearly γ was a flying node in T, otherwise a node greater than γ would appear in Γ.

To invert φ_d the outgoing edge of each flying node must be reverted to its original value in T. In order to correctly rebuild the path from 1 to 0, cycles of G must be considered in increasing order of their maximum node. □

From now on, vector p represents both the parent vector of the input tree and the function associated with the output digraph. The implementation of φ_d is as follows:

```
Algorithm: Compute $\varphi_d$
Input: an $n$-tree $T$ rooted in 0 represented by the parent vector $p$
Output: a functional digraph $G$ representing a function $p$
1. identify all flying nodes $f_1, f_2, \ldots, f_k$ as ordered in $1 \rightsquigarrow 0$.
2. for $i = 1$ to $k$ do
3.     swap $p(1)$ and $p(f_i)$.
4. return $G$ corresponding to $p$.
```

Compute φ_d requires linear time since Line 1 is a simple backwards scan of the path $1 \rightsquigarrow 0$. So the computation of the Dandelion code, when fitted into the General Scheme, is linear.

The decoding process is more difficult, but still can be done in linear time. Each node v in G belongs to a connected component that either contains a cycle or is a simple tree. Let us call γ_v the maximum node in the cycle of v's connected component or the root if the connected component is a tree. The hardest part in the decoding process is the computation of φ_d^{-1}: it consists in identifying the flying nodes. To this purpose the maximum node that can be reached from v must be computed, i.e., $\mu(v) = max\{p(v) \rightsquigarrow \gamma_v\}$. This can be done without explicitly identifying γ_v and 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 whether a node is still being processed or not. Initially *status*(v) = *unprocessed* for each node v. The computation of $\mu(v)$, for all nodes v, is detailed in the algorithm Compute φ_d^{-1} given in Box I.

Let us explain the condition of Line 6 in function ComputeMu. Since the status of a node v is *inProgress* only during a recursive call on the ascending path of v, when the condition of Line 6 is true a cycle has been identified: following outgoing edges ComputeMu moved from v back to v itself. At this point, again, infinite recursions must be avoided: an auxiliary function to compute the maximum in the cycle is called and the recursion terminates. The auxiliary function MaxInCycle simply scans outgoing edges starting from v until it reaches v and returns the maximum label. Line 17 requires a simple integer sorting; this ensures the linearity of the algorithm for computing φ_d^{-1}.

An example of φ_d and φ_d^{-1} concludes this section. Applying Compute φ_d to the tree T in Fig. 2(a) the following flying nodes are identified in the path $1 \rightsquigarrow 0 : f_1 = 10$ and $f_2 = 5$. The first iteration of the loop swaps $p(1)$ with $p(10)$ obtaining $p(1) = 2$ and $p(10) = 3$, the second one swaps $p(1)$ with $p(5)$ obtaining $p(1) = 0$ and $p(5) = 2$. The resulting functional digraph G is shown in Fig. 2(c). The codeword produced by the General Encoding Scheme from G (with $x = 0$ and $y = 1$) is $C = (5, 6, 10, 2, 4, 2, 1, 0, 3, 9)$, i.e., the Dandelion code corresponding to T.

When the General Decoding Scheme is used on $C = (5, 6, 10, 2, 4, 2, 1, 0, 3, 9)$ (with $x = 0$ and $y = 1$) a functional digraph equal to G in Fig. 2(c) is obtained. The procedure Compute φ_d^{-1} applied to G computes the following values: $\mu(0) = 0$, $\mu(1) = 0$, $\mu(2) = 5$, $\mu(3) = 10$, $\mu(4) = 10$, $\mu(5) = 5$, $\mu(6) = 10$, $\mu(7) = 5$, $\mu(8) = 1$, $\mu(9) = 0$, $\mu(10) = 10$, $\mu(11) = 9$. This implies $f_1 = 10$ and $f_2 = 5$. Finally the swaps of p values performed in the encoding are reverted. Then the resulting vector p describes the original T as shown in Fig. 2(a).

5. Dandelion-like codes

The class of Dandelion-like codes containing eight bijective codes was introduced in a paper on Genetic Algorithms [29]. In this paper the authors, analyzing the decoding phase of the Dandelion code, underlined that it is possible to introduce four changes into the procedure that builds a tree from a string. Among the sixteen potential codes induced by these four changes, only eight are bijective: the Dandelion-like codes.

In Table 2 these eight codes are described introducing just three changes into the encoding phase, in order to avoid discarding any code. This is a more natural way of describing the class of Dandelion-like codes and the three changes are as follows:

```
Algorithm: COMPUTE $\varphi_d^{-1}$
Input: a functional digraph $G$ representing a function $p$
Output: an $n$-tree $T$ represented by a parent vector $p$

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.          $\mu(v) = \text{MaxInCycle}(v)$.
8.      else
9.          ComputeMu($p(v)$).                          // recursive call
10.         $\mu(v) = max\{\mu(p(v)), p(v)\}$.
11.     $status(v) = processed$.
main
12. $\mu(0) = 0$.
13. $status(0) = processed$.
14. for $v = 1$ to $n-1$ do
15.     if $status(v) \neq processed$ then
16.         ComputeMu($v$).
17. identify all nodes $f_1, f_2, \ldots, f_k$ such that $\mu(f_i) = f_i$ in decreasing order.
18. for $i = k$ downto $1$ do
19.     swap $p(1)$ and $p(f_i)$.
20. return $T$ corresponding to $p$.
```

Box I.

Table 2
The 8 bijective Dandelion-like codes.

Code	Max/min	Up/down	Edge orientation
C_1	Max	Up	Preserve
C_2	Max	Down	Invert
C_3	Max	Down	Preserve
C_4	Max	Up	Invert
C_5	Min	Up	Preserve
C_6	Min	Down	Invert
C_7	Min	Down	Preserve
C_8	Min	Up	Invert

1. use minimum instead of maximum to compute flying nodes among those nodes in the path $1 \rightsquigarrow 0$;
2. search downward in the path from a node v to the node 1 instead of searching upward in the path $v \rightsquigarrow 0$;
3. invert the orientation of all edges in cycles.

As Paulden and Smith observed, C_1 is the Dandelion code, C_2 is the Happy code, C_3 is the MHappy code, and C_5 is the ϑ_n bijection.

The following sections introduce parameterized encoding and decoding algorithms suitable for all Dandelion-like codes. We later introduce parallel algorithms for these codes, but sequential versions appear first, in forms that are easy to parallelize though they may appear more complex than necessary.

5.1. Encoding algorithm

We formalize the three parameters required to characterize Dandelion-like codes as follows:

1. $\mu \in \{min, max\}$ specifies whether to search for minimum or maximum values;
2. $\Updownarrow \in \{up, down\}$ establishes if the μ values should be searched upward or downward;
3. INVERTEDGES is a boolean value that discriminates whether the orientation of cycle edges should be inverted or not;

For each v in $1 \rightsquigarrow 0$, the value $\mu_{\Updownarrow}(v)$ represents the maximum/minimum value above/below node v and $pred(v)$ represents the predecessor of v in the path $1 \rightsquigarrow 0$. Thus, depending on the parameters μ and $\Updownarrow$, the function $\mu_{\Updownarrow}(v)$ can be one of the followings:

$$\max_{\text{up}}(v) = \max\{w \in v \rightsquigarrow 0\} \qquad \max_{\text{down}}(v) = \max\{w \in 1 \rightsquigarrow v\}$$
$$\min_{\text{up}}(v) = \min\{w \in v \rightsquigarrow 0\} \qquad \min_{\text{down}}(v) = \min\{w \in 1 \rightsquigarrow v\}.$$

The encoding algorithm is as follows:

```
Algorithm: DANDELION-LIKE ENCODING ALGORITHM
Parameters: $\mu$, $\Updownarrow$, INVERTEDGES
Input: an $n$-tree $T$ rooted in 0 represented by the parent vector $p$
Output: a functional digraph $G$ representing a function $p$
1.  compute $\mu_{\Updownarrow}(v)$ for each $v$ in $1 \rightsquigarrow 0$.
2.  identify all nodes $f_1, f_2, \ldots, f_k$ such that $\mu_{\Updownarrow}(f_i) = f_i$.
3.  $f_0 = 1$;  $f_{k+1} = 0$.
4.  if $\Updownarrow = \text{down}$ then
5.      for $i = 1$ to $k$ do
6.          $f_i = pred(f_{i+1})$.
7.  for $i = 1$ to $k$ do
8.      swap $p(1)$ and $p(f_i)$.
9.  if INVERTEDGES then
10.     invert edges in all cycles.
11. return $G$ corresponding to $p$.
```

All values $\mu_{\Updownarrow}(v)$ can be computed in $O(n)$ time with simple forward/backwards scans of the path from 1 to 0; at the same time all f_i are identified. In Line 6, if $\Updownarrow=$ down, the highest node below f_{i+1} is identified, i.e., $pred(f_{i+1})$. Indeed, in this case, f_i should form a cycle with all nodes above it and below f_{i+1}. This operation can be performed in linear time traversing the path once again. Lines 7–8 intend to rearrange nodes in $1 \rightsquigarrow 0$ into cycles. The overall time complexity is linear since all operations can be accomplished traversing the path $1 \rightsquigarrow 0$ a limited number of times.

We conclude this section by showing the encoding of the tree in Fig. 3 with all Dandelion-like codes. The various codes identify the following nodes in Line 2:

$$C_1, C_4 : f_1 = 8, f_2 = 3 \qquad C_2, C_3 : f_1 = 4, f_2 = 6, f_3 = 8$$
$$C_5, C_8 : f_1 = 3 \qquad C_6, C_7 : f_1 = 4, f_2 = 3$$

thus introducing the following cycles in the functional digraph:

$$C_1, C_4 : (4, 6, 8), (3) \qquad C_2, C_3 : (4), (6), (8, 3)$$
$$C_5, C_8 : (4, 6, 8, 3) \qquad C_6, C_7 : (3, 6, 8), (4).$$

The resulting codewords are:

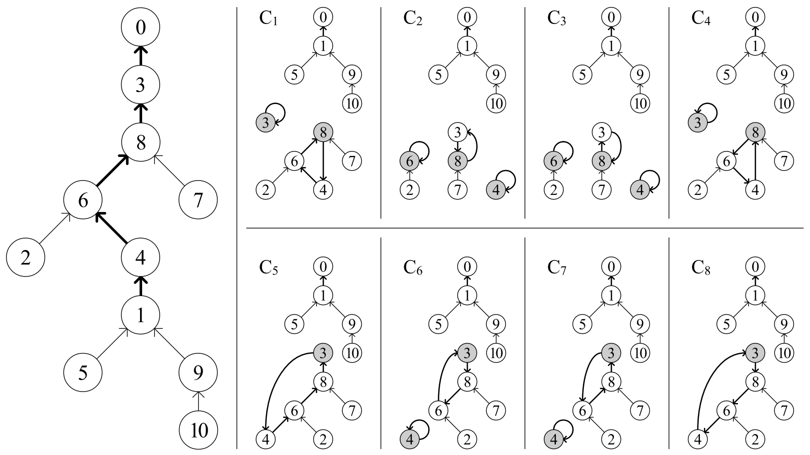

Fig. 3. A tree T of eleven nodes labeled from 0 to 10 and the 8 functional digraphs corresponding to the encoding of T with all Dandelion-like codes. Edges in the path $1 \rightsquigarrow 0$ in T (and the corresponding edges in all functional digraphs) are highlighted. Flying nodes are depicted in grey.

$C_1 : (6, 3, 6, 1, 8, 8, 4, 1, 9)$ $C_2 : (6, 8, 4, 1, 6, 8, 3, 1, 9)$
$C_3 : (6, 8, 4, 1, 6, 8, 3, 1, 9)$ $C_4 : (6, 3, 8, 1, 4, 8, 6, 1, 9)$
$C_5 : (6, 4, 6, 1, 8, 8, 3, 1, 9)$ $C_6 : (6, 8, 4, 1, 3, 8, 6, 1, 9)$
$C_7 : (6, 6, 4, 1, 8, 8, 3, 1, 9)$ $C_8 : (6, 8, 3, 1, 4, 8, 6, 1, 9)$.

Notice that the 8 codes are all different from each other, even though, on this small example, different codes produce the same codeword.

5.2. Decoding algorithm

The decoding algorithm identifies all cycles in a given functional digraph and computes, for each cycle C_i, the flying node f_i according with the function specified by the parameter μ. Afterwards, all cycles are broken and their nodes are placed in between 1 and 0 in such a way that the original path $1 \rightsquigarrow 0$ of the encoded tree is rebuilt. It is to underline that, before breaking cycles, the edge orientation should be reestablished if INVERTEDGES is true.

To reconstruct the correct order among the flying nodes, the algorithm considers the values of μ and $\Updownarrow$. If $\mu = \max$ and $\Updownarrow = \text{up}$ then the greater f_i must be below any other flying node, thus the f_i values must be ordered in decreasing order: $f_1 > f_2 > \cdots > f_k$. On the other hand, if $\Updownarrow = \text{down}$ the greater f_i must be placed above any other flying node, thus implying an increasing order: $f_1 < f_2 < \cdots < f_k$. If $\mu = \min$, the orders are reversed. So, the original path $1 \rightsquigarrow 0$ must be rebuilt according with the ordering of the f_i values:

$1 \rightsquigarrow f_1 \rightsquigarrow f_2 \rightsquigarrow \ldots \rightsquigarrow f_k \rightsquigarrow 0$.

Notice that all nodes in f_i's cycle must be placed right above or right below f_i depending on the value of $\Updownarrow$ (down or up, respectively). The computation of the reverse function is detailed in the DANDELION-LIKE DECODING ALGORITHM.

Algorithm: DANDELION-LIKE DECODING ALGORITHM
Parameters: μ, $\Updownarrow$, INVERTEDGES
Input: a functional digraph G representing a function p
Output: a tree T represented by its parent vector p

```
1.  find all cycles $C_i$ and their flying nodes $f_i$ according to $\mu$.
2.  if ($\mu = \max$ and $\Updownarrow = $ up) or ($\mu = \min$ and $\Updownarrow = $ down) then
3.    sort $\{f_i\}$ in decreasing order.
4.  else
5.    sort $\{f_i\}$ in increasing order.
6.  if INVERTEDGES then
7.    invert all edges in cycles.
8.  $f_0 = 1$; $f_{k+1} = 0$.
9.  if $\Updownarrow = $ down then
10.   for $i = 1$ to $k$ do
11.     $f_i = v_i \in C_i$ such that $p(v_i) = f_i$.
12. for $i = k$ downto 1 do
13.   swap $p(1)$ and $p(f_i)$.
```

Line 1 can be implemented in linear time as done for COMPUTE φ_d^{-1}. It is to underline that Line 11 (if required) identifies the only node v_i in the cycle of f_i such that $p(v_i) = f_i$: this node becomes the new flying node f_i. Since the computations of all v_i can be done in $O(n)$ time, the overall decoding procedure running time remains linear.

6. Parallel implementation

In this section the parallel implementation of the encoding and decoding algorithms proposed in Section 5 is detailed for the theoretical PRAM model.

We choose the classical PRAM 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 considered too abstract compared with actual parallel architectures. Due to the recent technological advances this trend is changing: Wen and Vishkin reported about the advancements achieved at the University of Maryland within the project PRAM-On-Chip [39]. 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 primitives like: Prefix-Sum, Join, Fetch and Increment, etc.) with the intent of providing an easy programming tool for implementing PRAM algorithms. Thus we think that PRAM is robust, reasonable, and well studied theoretical framework for describing high level parallel algorithms. For more information we refer to [37].

In the following some basic parallel techniques used in the parallelization of Dandelion-like encoding and decoding algorithms are listed. Details, proofs, and prerequisites can be found in the literature [12,23,35]. PRAM with Exclusive Write (EW) and both Exclusive Read (ER) and Concurrent Read (CR) are considered.

6.1. Basic parallel techniques

Scheduling principle: If a PRAM algorithm A' runs in time t using p' processors, then for any $p'' < p'$, there is an algorithm A'' for the same problem and the same PRAM model that runs in time $O(tp'/p'')$ with p'' processors.

Broadcast: On an EREW PRAM it is possible to broadcast a value x, held by a single processor, to all processors in $O(\log n)$ time using $O(n/\log n)$ processors.

Euler tour: An *Euler tour* is a cycle in a directed graph that traverses each edge exactly once. A free tree is converted into a directed graph by replacing each undirected edge $\{u, v\}$ by two directed edges (u, v) and (v, u). An Euler tour of a tree with n nodes can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM. The Euler tour can be used to root an unrooted tree and to optimally compute the level of each node.

Let $*$ be an associative binary operation over domain $\mathcal{D}$ that can be evaluated in $O(1)$ time using a single EREW PRAM processor.

Parallel Prefix-Sum: The *Parallel Prefix-Sum Problem* consists in computing, for all $j \in [1, n]$, the prefix sums $\Sigma_j = x_1 * \cdots * x_j$ where $x_i \in \mathcal{D}$ for $1 \leq i \leq n$. The Parallel Prefix-Sum Problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Parallel tree contraction: Let T_R be a regular binary tree with n nodes (i.e., a tree in which every internal node has exactly two children). Let its leaves be labeled by operands over domain $\mathcal{D}$ and its internal nodes be labeled by $*$. All the algebraic expressions associated with the internal nodes (one per node) of T_R can be evaluated in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

6.2. Parallel encoding

The implementation on an EREW PRAM of the encoding algorithm is the following:

```
Algorithm: Dandelion-Like Parallel Encoding Algorithm
Parameters: $\mu$, $\Updownarrow$, invertEdges
Input: a tree $T$ rooted in 0 represented by its parent vector $p$
Output: a functional digraph $G$ representing a function $p$
 1. compute $\min(T_v)$ and $level(v)$ for each $v \in T$.
 2. create $P$ corresponding to the path $1 \rightsquigarrow 0$.
 3. compute $\mu_{\Updownarrow}(v)$ for each $v \in P$ excluding 1 and 0.
 4. identify all nodes $f_1, f_2, \ldots, f_k$ such that $\mu_{\Updownarrow}(f_i) = f_i$.
 5. $f_0 = 1$; $f_{k+1} = 0$.
 6. if $\Updownarrow = \text{down}$ then
 7.    for $i = 1$ to $k$ in parallel do
 8.       $f_i = pred(f_{i+1})$.
 9. for $i = 1$ to $k$ in parallel do
10.    $p(f_i) = p(f_{i-1})$.
11. $p(1) = 0$.
12. if invertEdges then
13.    for each $v \in P$ in parallel do
14.       $p(p(v)) = v$.
```

Initially, the minimum value in the subtree rooted at v, $\min(T_v)$, is computed for each node v. If $\min(T_v) = 1$ then v is in the path $1 \rightsquigarrow 0$. This operation requires $O(\log n)$ time with $O(n/\log n)$ processors by using the Parallel Tree Contraction technique since min is a binary associative operation. It is to remark that it is always possible to convert a rooted tree T into a regular binary tree T_R. The construction replaces every node u in T having d children, $v_1, v_2, \ldots, v_d$, by $d+1$ nodes $u^1, u^2, \ldots u^{d+1}$. In T_R, u^{i+1} is the right child of u^i. If node v_i is the i-th child of u in T, then v_i^1 is the left child of u^i in T_R (see Fig. 4). This transformation can be obtained in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM [21].

With the same bounds we can compute the top-down level of each node (with the Euler Tour technique): exploiting this information we are able to create a vector P containing the sequence of all nodes in the path $1 \rightsquigarrow 0$.

The function $\mu_{\Updownarrow}(v)$ can be computed for all nodes in P (with the same time and processors bounds of the above operations) regarding this path as a tree T_P and computing the max/min value in the subtree of each node. If $\Updownarrow = \text{down}$ then T_P must be rooted in 0, otherwise it must be rooted in 1. To order the f_i values it is sufficient to enumerate them within vector P. This enumeration can be achieved with Prefix-Sum computation on an auxiliary

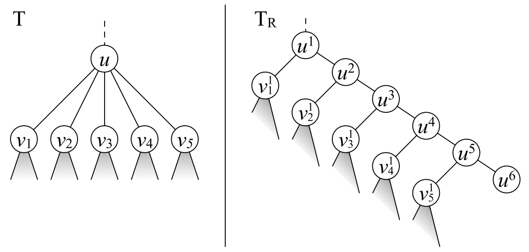

Fig. 4. A general tree T can be converted into a regular binary tree T_R replacing each node u with d children by $d+1$ nodes.

vector P' whose i-th element is equal to 1 if $\mu_{\Updownarrow}(P(i)) = P(i)$ and 0 otherwise.

The three loops in Lines 7, 9, and 13 require $O(1)$ time with n processors and do not imply concurrent reading or writing. The value $pred(v)$ (the predecessor of v in the path $1 \rightsquigarrow 0$) can be computed in the following way: for each node in $v \in P$ set $pred(p(v)) = v$. Applying the scheduling principle all these operations can be executed on $O(n/\log n)$ processors in $O(\log n)$ time. The overall cost (i.e., time multiplied number of processors) is linear on an EREW PRAM and thus the parallel algorithm is optimal since its cost matches the optimal sequential time.

6.3. Parallel decoding

The most demanding step in the parallel decoding algorithm is the computation of flying nodes. These nodes can be identified in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM in a Pointer Jumping like fashion: for each node the outgoing edge is followed searching for the max/min value in the ascending path (the parameter μ discriminates whether to search for maximum or minimum values). After each step we set $p(v) = p(p(v))$, thus obtaining a single Pointer Jump. The procedure is as follows:

```
Algorithm: Parallel Identification of Flying Nodes
 1. $p(0) = 0$.
 2. for each node $v \in T$ in parallel do
 3.    asc($v$) = $v$;
 4.    self($v$) = true.
 5. for $j = 1$ to $\lceil \log n \rceil$ do
 6.    for each node $v \in T$ in parallel do
 7.       if asc($p(v)$) = $\mu$\{asc($p(v)$), asc($v$)\} then
 8.          asc($v$) = asc($p(v)$);
 9.          self($v$) = false.
10.       $p(v) = p(p(v))$.
```

The value asc(v) is the min/max value in the ascending path of v. At each step asc(v) is compared with asc($p(v)$) and updated, if it is the case. Notice that the algorithm explicitly flag whether the value asc(v) comes from v itself or has been encountered in the proper ascending path. At the end of the $\log n$ iterations if asc(v) = v and self(v) is false, v is the min/max node in its cycle. Indeed, a value equal to v has been found in the proper ascending path of v, this means that there exists a path $v \rightsquigarrow v$, i.e., a cycle. Moreover, no node smaller/greater that v has been encountered in this cycle. All these nodes can be enumerated with Prefix-Sum to generate the (increasing or decreasing) ordered sequence of the flying nodes $f_1, f_2, \ldots, f_k$. A copy of p should be used in the Parallel Identification of Flying Nodes, since the original vector p will be required latter in the decoding.

Once flying nodes have been identified, a Broadcast operation can be used to propagate a flag in their ascending paths, thus identifying all nodes belonging to any cycle. The decoding algorithm proceeds as follows:

Table 3
Costs of known and new algorithms for bijective string-based codes. Costs of parallel algorithms are expressed as the number of processors multiplied by the maximum time required by a single processor. Algorithms marked with CR require concurrent read, while all other algorithms work on an EREW PRAM.

		Sequential		Parallel	
		Encoding	Decoding	Encoding	Decoding
Prüfer-like	Prüfer	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$
	2nd Neville	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$	$O(n\sqrt{\log n})$
	3rd Neville	$O(n)$	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$
	Stack-Queue	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$	$O(n\sqrt{\log n})$
	Chen	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$
Dandelion-like	Dandelion	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$ CR
	ϑ_n bijection	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$ CR
	Happy	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$ CR
	MHappy	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$ CR
	Blob	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$ CR

Algorithm: Dandelion-Like Parallel Decoding Algorithm
Parameters: μ, $\Updownarrow$, invertEdges
Input: a functional digraph G represented by a vector p
Output: a tree T corresponding to the parent vector p

```
1. identify all flying nodes $f_1, f_2, \ldots, f_k$ according with $\mu$ and $\Updownarrow$.
2. if invertEdges then
3.   for each $v \in$ cycles in parallel do
4.     $p(p(v)) = v$.
5. $f_0 = 1$; $f_{k+1} = 0$.
6. if $\Updownarrow$ = down then
7.   for $i = 1$ to $k$ in parallel do
8.     $f_i = pred(f_i)$.
9. for $i = 0$ to $k-1$ in parallel do
10.   $p(f_i) = p(f_{i+1})$.
11. $p(f_k) = 0$.
```

All the three loops in Lines 3, 7, and 9 require $O(1)$ time with $O(n)$ processors on an EREW PRAM, provided that the *pred* values are computed for all nodes belonging to any cycle (as described in the parallel encoding for nodes in P). The overall cost of Dandelion-Like Parallel Decoding Algorithm on a CREW PRAM is $O(n \log n)$, due to the bottleneck of this algorithm that is the Parallel Identification of Flying Nodes.

7. Conclusions and open problems

This paper presents what, to the best of our knowledge, are the first encoding and decoding parallel algorithms for the Dandelion-like codes. A unique encoding algorithm and a unique decoding algorithm have been designed which, properly parameterized, can be used for all eight codes. These algorithms, optimal in the sequential setting, have been implemented on the classical PRAM model.

The encoding algorithm has been parallelized using an EREW PRAM with $O(n/\log n)$ processors and requires $O(\log n)$ time: the overall cost is consequently linear and the algorithm is optimal. Concerning Prüfer-like codes, we remark that for the Second Neville code and for the Stack-Queue code parallel encoding is not optimal: a bottleneck arises from the use of an integer sorting algorithm. This bottleneck is avoided for Dandelion-like codes since all the sorting requirements are accomplished with simple Prefix-Sum computations.

The decoding algorithm can be efficiently parallelized on a CREW PRAM to run in $O(\log n)$ time with $O(n)$ processors: the overall cost is $O(n \log n)$. The only non-optimal step of the parallel implementation is the computation of characteristic nodes in cycles realized in a Pointer Jumping like fashion. It does not seem possible to avoid this step, but it remains an open problem to understand if this computation can be achieved with linear cost. Table 3 completes Table 1 with the results presented in this paper. Observing these results it is clear that the problem of designing optimal parallel algorithms is still open for several codes. In particular no optimal decoding algorithm is known: the best decoding algorithms cost $O(n\sqrt{\log n})$ on an EREW PRAM. It remains an interesting open problem to design a bijective code that admits an optimal parallel decoding algorithm, or to show that such a result cannot be achieved by providing a lower bound to the parallel cost of decoding.

References

[1] Y.T. Bau, C.K. Ho, H.T. Ewe, Ant colony optimization approaches to the degree-constrained minimum spanning tree problem, Journal of Information Science and Engineering 24 (2008) 1081–1094.
[2] V. Boppana, I. Hartanto, W.K. Fuchs, Full fault dictionary storage based on labeled tree encoding, in: Proceedings of the 14th IEEE VLSI Test Symposium, VTS'96, 1996, pp. 174–179.
[3] S. Caminiti, On Coding Labeled Trees, Ph.D. Thesis, Sapienza University of Rome, 2007.
[4] S. Caminiti, N. Deo, P. Micikevičius, Linear-time algorithms for encoding trees as sequences of node labels, Congressus Numerantium 183 (2006) 65–75.
[5] S. Caminiti, I. Finocchi, R. Petreschi, On coding labeled trees, Theoretical Computer Science 382 (2) (2007) 97–108 (Special issue devoted to the best papers of LATIN'04).
[6] S. Caminiti, R. Petreschi, Optimal algorithms for Chen code, Technical Report TR-02-2009, Department of Computer Science, Sapienza University of Rome, March 2009.
[7] S. Caminiti, R. Petreschi, String coding of trees with locality and heritability, in: Proceedings of the 11th International Conference on Computing and Combinatorics, COCOON'05, in: LNCS, vol. 3595, 2005, pp. 251–262.
[8] S. Caminiti, R. Petreschi, Parallel algorithms for Blob code, in: Proceedings of the 3rd Workshop on Algorithms and Computation, WALCOM'2010, in: LNCS, vol. 5942, 2010, pp. 167–178.
[9] S. Caminiti, R. Petreschi, Parallel algorithms for Dandelion-Like codes, in: Proceedings of the 9th International Conference on Computational Science, ICCS'09, in: LNCS, vol. 5544, 2009, pp. 611–620.
[10] A. Cayley, A theorem on trees, Quarterly Journal of Mathematics 23 (1889) 376–378.
[11] W.Y.C. Chen, A general bijective algorithm for trees, in: Proceeding of the National Academy of Science, vol. 87, 1990, pp. 9635–9639.
[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, McGraw-Hill, 2001.
[13] N. Deo, N. Kumar, V. Kumar, Parallel generation of random trees and connected graphs, Congressus Numerantium 130 (1998) 7–18.
[14] N. Deo, 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 (2002) 77–85.
[15] N. Deo, P. Micikevičius, Prüfer-like codes for labeled trees, Congressus Numerantium 151 (2001) 65–73.
[16] N. Deo, 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.
[17] L. Devroye, Non-Uniform Random Variate Generation, Springer-Verlag, New York, 1986.
[18] W. Edelson, M.L. Gargano, Feasible encodings For GA solutions of constrained minimal spanning tree problems, in: Proceedings of the 5th Genetic and Evolutionary Computation Conference, GECCO'00, Las Vegas, NV, USA, 2000, pp. 82–89.
[19] Ö. Eğecioğlu, J.B. Remmel, Bijections for Cayley trees, spanning trees, and their q-analogues, Journal of Combinatorial Theory 42A (1) (1986) 15–30.
[20] V.K. Garg, A. Agarwal, Distributed maintenance of a spanning tree using labeled tree encoding, in: Proceedings of 11th International Euro-Par Conference, Euro-Par'05, in: LNCS, vol. 3648, 2005, pp. 606–616.
[21] R. Greenlaw, M.M. Halldórsson, R. Petreschi, On computing Prüfer codes and their corresponding trees optimally in parallel, in: Proceedings of Journées de l'Informatique Messine, JIM'00, Metz, France, 2000.

[22] R. Greenlaw, R. Petreschi, Computing Prüfer codes efficiently in parallel, Discrete Applied Mathematics 102 (3) (2000) 205–222.
[23] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[24] P. Klingsberg, Doctoral Dissertation, Ph.D. Thesis, University of Washington, Seattle, Washington, 1977.
[25] G. Kreweras, P. Moszkowski, Tree codes that preserve increases and degree sequences, Journal of Discrete Mathematics 87 (3) (1991) 291–296.
[26] P. Micikevičius, Parallel graph algorithms for molecular conformation and tree codes, Ph.D. Thesis, University of Central Florida, 2002.
[27] J.W. Moon, Counting Labeled Trees, William Clowes and Sons, London, 1970.
[28] E.H. Neville, The codifying of tree-structure, in: Proceedings of Cambridge Philosophical Society, vol. 49, 1953, pp. 381–385.
[29] T. Paulden, 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, CEC'06, 2006, pp. 2111–2118.
[30] Y.A. Phoulady, M. Behzadi, H. Taheri, Sharing a labeled tree, in: Proceedings of the 4th Benelux Workshop on Information and System Security, WISSec'09, 2009.
[31] S. Picciotto, How to Encode a Tree, Ph.D. Thesis, University of California, San Diego, 1999.
[32] H. Prüfer, Neuer Beweis eines Satzes über Permutationen, Archiv der Mathematik und Physik 27 (1918) 142–144.
[33] P. Rao, B. Moon, Prix: indexing and querying xml using prufer sequences, in: Proceedings of the International Conference on Data Engineering, ICDE'04, 2004, pp. 288–300.
[34] C.R. Reeves, J.E. Rowe, Genetic Algorithms: A Guide to GA Theory, Springer, 2003.
[35] J.H. Reif, Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
[36] C. Vanniarajan, K. Krithivasan, Network (tree) topology inference based on Prüfer sequence, in: Proceedings of the 16th National Conference on Communications, NCC'10, 2010.
[37] U. Vishkin, G.C. Caragea, B. Lee, Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform, in: S. Rajasekaran, J. Reif (Eds.), Handbook of Parallel Computing: Models, Algorithms and Applications, CRC Press, 2008 (Chapter 5).
[38] Yue-Li Wang, Hon-Chan Chen, Wei-Kai Liu, A parallel algorithm for constructing a labeled tree, IEEE Transactions on Parallel and Distributed Systems 8 (1997) 1236–1240.
[39] X. Wen, U. Vishkin, PRAM-on-chip: first commitment to silicon, in: Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures, SPAA'07, 2007, pp. 301–302.

Saverio Caminiti graduated in computer science in 2003 and obtained his Ph.D. degree in computer science in 2008, from Sapienza University of Rome, Italy. He currently has a post-doctoral position. His research interests include graph algorithms, combinatorial mappings, layout of network topologies.

Rossella Petreschi graduated in mathematics in 1972. From 1973 to 1988 she was, first, with the Italian National Research Council, then with the Department of Mathematics at University of L'Aquila, Italy, and finally with the Department of Mathematics at Sapienza University of Rome. Since its constitution in 1989, she has been with the Department of Computer Science at Sapienza University of Rome, where she is currently a full professor in computer science. Her current research interests include the design of sequential and parallel algorithms to solve problems, arising from the project of computer science systems, that find natural description in terms of graph theory.