Parallel Algorithms for Dandelion-Like Codes*

Saverio Caminiti and Rossella Petreschi

Computer Science Department, *Sapienza* University of Rome
Via Salaria, 113 - I00198 Rome, Italy
{caminiti,petreschi}@di.uniroma1.it

Abstract. We consider the class of Dandelion-like codes, i.e., a class of bijective codes for coding labeled trees by means of strings of node labels. In the literature it is possible to find optimal sequential algorithms for codes belonging to this class, but, for the best of our knowledge, no parallel algorithm is reported. In this paper we present the first encoding and decoding parallel algorithms for Dandelion-like codes. Namely, we design a unique encoding algorithm and a unique decoding algorithm that, properly parametrized, can be used for all Dandelion-like codes. 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). Many are the usual representations of tree data structures: adjacency matrices, adjacency lists, parent vectors, and balanced parentheses are just a few examples. An interesting alternative 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 [1], distributed spanning tree maintenance [2], generation of random trees [3], Genetic Algorithms [4]. In this paper we restrict our attention to bijective string-based codes in which the length of the string must be equal to $n-2$ [5] (n is the number of nodes of the encoded tree). The first bijective string-based coding for trees is due to Prüfer [6]. since then, many other bijective codes have been introduced [7,8,9,10,11,12,13,14].

Concerning algorithmic aspects of these codes, there is a wide literature presenting optimal sequential algorithms for encoding and decoding all of them. In particular we refer to [15] for the class of Prüfer-like codes (the first 4 codes in Table 1)

* Work partially supported by *Sapienza* University of Rome under the project "Strutture Dati e Tecniche Algoritmiche Evolute per Modelli di Calcolo Innovativi".

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 611–620, 2009.

Table 1. Costs of known algorithms for bijective string-based codes. Parallel costs are expressed as the number of processors × maximum execution time.

	Sequential		Parallel	
	Encoding	Decoding	Encoding	Decoding
Prüfer [6]	$O(n)$	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$
2nd Neville [13]	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$	$O(n\sqrt{\log n})$
3rd Neville [13]	$O(n)$	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$
Stack-Queue [9]	$O(n)$	$O(n)$	$O(n\sqrt{\log n})$	$O(n\sqrt{\log n})$
θ_n [10]	$O(n)$	$O(n)$	unknown	unknown
Happy [14]	$O(n)$	$O(n)$	unknown	unknown
Dandelion [14]	$O(n)$	$O(n)$	unknown	unknown
MHappy [7]	$O(n)$	$O(n)$	unknown	unknown
Blob [11,14]	$O(n)$	$O(n)$	unknown	unknown
Chen [8]	$O(n)$	$O(n)$	unknown	unknown

and to [7] for the other codes. A survey of optimal sequential algorithms can be found in [16]. Also parallel algorithms have been studied [7,17,18,19], but results are known only for Prüfer-like codes. More results related to sequential and parallel algorithms for bijective codes can be found in [20] and are summarized in Table 1.

In this paper we make a further step in understanding the feasibility of encoding and decoding in a parallel setting. We focus on the class of Dandelion-like codes introduced in [21] and we present efficient parallel algorithms for encoding and decoding all these codes. This class contains the second block of codes (rows 5 to 8) in Table 1. Dandelion-like codes are especially useful in Genetic Algorithms (GA) since experimental analysis show that Prüfer-like codes perform poorly (with respect to GA requirements) [22] while Dandelion-like codes achieve best results [21]. The techniques we use for Dandelion-like codes can be also exploited to obtain efficient parallel algorithms for encoding and decoding the Blob code (rows 5 in Table 1).

The paper is organized as follows: after a few preliminary definitions, in Sec.3 we recall the Dandelion code, together with examples. In Sec.4 we recall the class of Dandelion-like codes and introduce an encoding and a decoding algorithm that, properly parametrized, can be used for all codes in the class. In a sequential setting these two algorithms run in linear time and are therefore optimal. In Sec.6 we discuss how to parallelize our algorithms for the PRAM model. For the encoding algorithm we obtain optimal linear cost implementation on an EREW PRAM. The decoding algorithm requires $O(n \log n)$ cost on a CREW PRAM.

2 Preliminary Definitions

In this section, we introduce some definitions that will be useful in the rest of this paper. As usual, the notation $[a, b]$ represents the integer interval from a to b, both included.

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

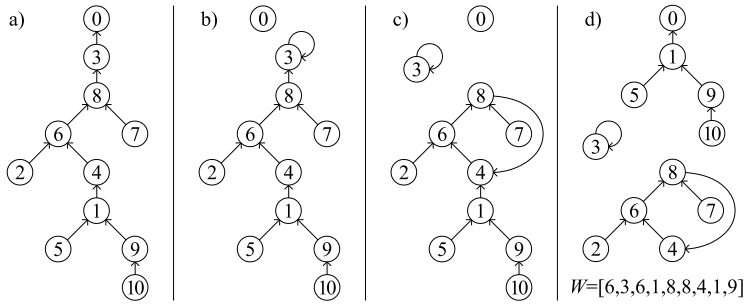

Fig. 1. Example of Dandelion encoding algorithm execution

Since in this paper we only deal with labeled trees, we will refer to them simply as trees. In the following all trees will be regarded as rooted in the fixed node 0 and its edges will be oriented upward from a node to its parent; an example is depicted in Fig.1a.

Definition 2. *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 $V = \{0, \ldots, n\}$ and $E = \{(v, g(v))$ for every $v \in V\}$.*

For this class of graphs the following lemma holds:

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

Functional digraphs are easily generalizable for representing functions which are undefined in some values: if $g(x)$ is not defined, the node x in G does not have outgoing edges.

As an example consider a rooted tree T and let $p[v]$ be the parent of v for each v in T. T is the functional digraph associated with the function p.

In the following, when no confusion arise, we will consider vectors as functions and vice versa. The notation $u \rightsquigarrow v$ identifies the directed path from node u to node v; $u \rightsquigarrow u$ is the degenerate path of length 0. Loops will be considered as cycles of length 1.

3 The Dandelion Code

We now recall the Dandelion code (originally introduced in [14]) as reinterpreted in [7] in which a tree T is transformed into a functional digraph G_g. Initially $G_g = T$, thus the function g is equivalent to the parent vector of T. The Dandelion code choose node 1 to play a special role and rearrange all nodes in $1 \rightsquigarrow 0$ into cycles; the resulting digraph will correspond to a function g such that $g[0]$ is undefined and $g[1] = 0$. Let us detail the rearrangement of nodes in $1 \rightsquigarrow 0$:

Let $1 = v_1, v_2, \ldots, v_l = 0$ be all nodes in $1 \rightsquigarrow 0$ and let $m_i = \max\{v_i, \ldots, v_m\}$. Among them we choose all nodes such that $m_i = v_i$ (excluding v_l); these nodes

identify a subsequence $f_1, f_2, \ldots, f_k$ of sequence $v_1, v_2, \ldots, v_{l-1}$. We assign $f_0 = 1$. As an example consider the tree of 11 nodes, labeled from 0 to 10 in Fig.1a in which the path $1 \rightsquigarrow 0$ is (1, 4, 6, 8, 3, 0). We obtain $f_0 = 1$, $f_1 = 8$, $f_2 = 3$. The algorithm proceeds by set $g[f_i] = g[f_{i-1}]$ for each i from k down to 1. During this process all nodes in the original path between 1 and 0 are partitioned into several cycles (or loops). At the end the path $1 \rightsquigarrow 0$ is reduced to a single edge $(1, 0)$ and the resulting codeword is $g[2], g[3], \ldots, g[n-1]$. In our example, since $f_0 = 1$, $f_1 = 8$, and $f_2 = 3$, the algorithm set $g[3] \leftarrow g[8] = 3$, introducing a loop, and $g[8] \leftarrow g[1] = 4$, introducing a cycle. Fig.1d shows the resulting digraph: the associated codeword is $W = [6, 3, 6, 1, 8, 8, 4, 1, 9]$.

The procedure can be easily inverted to obtain the tree T corresponding to any given codeword $W = w_1, w_2, \ldots, w_{n-2}$. Initially reconstruct the functional digraph corresponding to g defined as: $g[0]$ is undefined, $g[1] = 0$, and $g[i] = w_{i-1}$. Then, identify all cycles $C_1, C_2, \ldots, C_k$ in G_g. For each cycle C_i, let us call *characteristic* its maximum node $f_i = \max\{v \in C_i\}$. W.l.o.g. assume that characteristic nodes are numbered such that $f_1 > f_2 > \ldots > f_k$.

The reconstruction of the original path $1 \rightsquigarrow 0$ is obtained reinserting all nodes of each cycle in between 1 and 0. Cycles are selected in descending order with respect to their characteristic values and, for each i from 0 to $k-1$, we set $g[f_i] = g[f_{i+1}]$ (where $f_0 = 1$). At the end we set $g[f_k] = 0$: G_g is now equal to the tree T corresponding to the given codeword W.

As an example of decoding consider the codeword $W = [6, 3, 6, 1, 8, 8, 4, 1, 9]$. It is easy to see that, according with the reconstruction described above, we initially obtain the functional digraph represented in Fig.1d. This graph has two cycles: one is a loop on node 3, and one is induced by nodes 6, 8, and 4. Thus we obtain $f_0 = 1$, $f_1 = 8$, $f_2 = 3$. Then we set $g[1] \leftarrow g[8] = 4$, $g[8] \leftarrow g[3] = 3$, and $g[3] \leftarrow 0$. Now G_g is exactly the tree represented in Fig.1a.

4 Dandelion-Like Codes

In this section we report the class of Dandelion-like codes introduced in [21]. We describe them in terms of functional digraphs in the style of [7] and we explicitly present one optimal encoding and one optimal decoding parametrized algorithms that can be used for all these codes.

Looking at the Dandelion code presented in Sec.3 it is easy to see that several details can be changed to originate different codes:

1. use minimum instead of maximum to compute f_i nodes among those in the path $1 \rightsquigarrow 0$ (the characteristic nodes in the decoding procedure);
2. search downward in the path from f_i to 1 instead of searching upward in the path from f_i to 0;
3. invert the orientation of all edges in cycles.

A summary of the 2^3 bijective codes obtained by all possible combinations of the three changes is reported in the following table – it is to notice that in [21] the 8 bijective codes reported were selected among 16 codes generated considering 4 possible changes of the decoding phase. Codes are numbered according with [21].

Code	max/min	up/down	edge orientation	Code name
C_1	max	up	preserve	Dandelion [14]
C_2	max	down	invert	Happy [14]
C_3	max	down	preserve	MHappy [7]
C_4	max	up	invert	
C_5	min	up	preserve	θ_n bijection [10]
C_6	min	down	invert	
C_7	min	down	preserve	
C_8	min	up	invert	

Encoding Algorithm. In this section we present an algorithm able to encode all the 8 Dandelion-like codes. In Program 1 we report only the fragment of code that transforms a tree into a functional digraph: obtaining the codeword from the functional digraph and vice versa is straightforward (see Sec.3). The DANDELION-LIKE ENCODING ALGORITHM has three parameters:

1. $\mu \in \{min, max\}$ specifies whether to search for minimum or maximum values;
2. $\uparrow\downarrow \in \{up, down\}$ 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.

The value $\mu_{\uparrow\downarrow}(v)$ represents, for each v in the path $1 \rightsquigarrow 0$, the maximum/minimum value above/below node v (including v itself). Thus, depending on the parameters μ and $\uparrow\downarrow$, the function $\mu_{\uparrow\downarrow}(v)$ can be one of the followings:

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

Let us now analyze the algorithm. All values $\mu_{\uparrow\downarrow}(v)$ can be computed with simple forward/backward scan of the path $1 \rightsquigarrow 0$; at the same time all f_i are identified. This requires $O(n)$ time. In line 4, if $\uparrow\downarrow$ = down, the highest node before f_{i+1} is identified. Indeed, in this case, f_i should form a cycle with all node above it and below f_{i+1}. This operation can be performed in linear time traversing the path $1 \rightsquigarrow 0$ once again. The same time bound holds for lines 5–6. Line 7 can efficiently be implemented in the following way: each node v in the path $1 \rightsquigarrow 0$ sets itself as the new parent of its current parent (i.e., $p'[p[v]] = v$) and all values in p are updated according with the values in p'. The overall time complexity of the DANDELION-LIKE ENCODING ALGORITHM is linear.

Consider, as an example, the encoding of the tree shown in Fig.1a with all possible 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 G_p:

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

The resulting codewords are: 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],

Program 1. Dandelion-Like Encoding Algorithm

Parameters: μ, $\uparrow\downarrow$, INVERTEDGES
Input: a tree T represented by its parent vector p
Output: a functional digraph G_p such that $p[0] = undef$ and $p[1] = 0$

```
1. Compute μ_{↑↓}(v) for each v in 1 ⇝ 0 excluding 1 and 0
2. Identify all nodes f_1, f_2, ..., f_k such that μ_{↑↓}(f_i) = f_i
3. f_0 = 1; f_{k+1} = 0
4. if ↑↓ = down then f_i = {v ∈ 1 ⇝ 0 : p[v] = f_{i+1}}   ∀ 1 ≤ i ≤ k
5. for i = k down to 1 do p[f_i] = p[f_{i-1}]
6. p[1] = 0
7. if INVERTEDGES then invert all edges in cycles
```

Program 2. Dandelion-Like Decoding Algorithm

Parameters: μ, $\uparrow\downarrow$, INVERTEDGES
Input: a functional digraph G_p such that $p[0] = undef$ and $p[1] = 0$
Output: a tree T represented by its parent vector p

```
1. Find all cycles C_i and their characteristic nodes f_i according to μ
2. if (μ = max and ↑↓ = up) or (μ = min and ↑↓ = down) then
3.    Sort {f_i} in decreasing order
4. else Sort {f_i} in increasing order
5. if INVERTEDGES then invert all edges in cycles
6. f_0 = 1; f_{k+1} = 0
7. if ↑↓ = down then f_i = {v ∈ C_i : p[v] = f_i}   ∀ 1 ≤ i ≤ k
8. for i = 0 to k - 1 do p[f_i] = p[f_{i+1}]
9. p[f_k] = 0
```

$C_5 = [6, 4, 6, 1, 8, 8, 3, 1, 9]$, $C_6 = [6, 3, 8, 1, 4, 8, 6, 1, 9]$, $C_7 = [6, 3, 6, 1, 8, 8, 4, 1, 9]$, and $C_8 = [6, 8, 3, 1, 4, 8, 6, 1, 9]$.

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

Decoding Algorithm. We now describe how to invert the transformation: given a functional digraph we identify its cycles and compute, for each cycle C_i the characteristic node f_i according with the function specified by the parameter μ. Then 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 tree is reconstructed. If INVERTEDGES is true then, before breaking cycles, the edge orientation should be reestablished.

Let us now describe how to reconstruct the correct order among the characteristic nodes. If $\mu = \max$ and $\uparrow\downarrow = \text{up}$ then greater f_i must be below any other characteristic node, thus the f_i values must be ordered in decreasing order: $f_1 > f_2 > \ldots > f_k$. On the other hand, if $\uparrow\downarrow = \text{down}$ then the greater f_i must be placed above any other characteristic node, thus implying an increasing order: $f_1 < f_2 < \ldots < f_k$. If $\mu = \min$ the orders are reversed. So, the path $1 \rightsquigarrow 0$ have to be rebuilt according with the ordering of the f_i: $1 \rightsquigarrow f_1 \rightsquigarrow \ldots \rightsquigarrow f_k \rightsquigarrow 0$.

Finally notice that, if $\uparrow\downarrow = \text{up}$ then the characteristic node should be above all nodes of its cycle, otherwise it should be below them. This is all we need to correctly rebuild the path $1 \rightsquigarrow 0$ of the original encoded tree.

Program 3. Identification of Characteristic Nodes

function Analyze(v)

```
1. status(v) = inProcess
2. if status(p[v]) = inProgress then compute μ value in cycle p[v] ⇝ v
3. else if status(p[v]) ≠ processed then Analyze(p[v])
4. status(v) = processed
```

main

```
1. for v = 2 to n − 1 do
2.    if status(v) ≠ processed then Analyze(v)
```

The computation of the reverse function is detailed in Program 2. Line 1 can be implemented by means of a recursive function that follows the outgoing edge of each node until it identifies a cycle, then an auxiliary function is used to compute the min/max value in that cycle (see Program 3). This requires $O(n)$ time. An integer sorting algorithm can be used to sort the f_i values in increasing or decreasing order and cycles edges can be inverted as described in the analysis of the encoding algorithm. Line 7 (if required) identifies the only node v in the cycle of f_i such that $p[v] = f_i$, each such node becomes new value for f_i: this require $O(n)$ time. Thus, the overall decoding procedure requires linear time.

5 Parallel Implementation

In this section we present a parallel version of the encoding and decoding algorithms proposed in Sec.4. Our algorithms are described for the theoretical PRAM model and costs are expressed as the number of processors multiplied by the maximum time required by a single processor.

We choose the PRAM classical 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. As noted in the introduction, this trend is changing. At SPAA'07, Vishkin and Wen reported about the recent advancements achieved at the University of Maryland within the project PRAM-On-Chip [23]. 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 providing an easy programing tool to implement PRAM algorithms. It has primitives like: Prefix Sum, Join, Fetch and Increment, etc. Thus we think that PRAM is robust, reasonable, and well studied theoretical framework for describing high level parallel algorithms.

In the following we will consider PRAM with Exclusive Write (EW) and either Exclusive Read (ER) or Concurrent Read (CR). Due to the lack of space, we don't explicitly recall the well known basic techniques used in this section; we refer the interested reader to [24].

Parallel Encoding. We now describe how to encode Dandelion-like codes on an EREW PRAM; the parallel algorithm is detailed in Program 4.

Program 4. Dandelion-Like Parallel Encoding Algorithm

Parameters: μ, $\Uparrow\!\downarrow$, INVERTEDGES
Input: a tree T represented by its parent vector p
Output: a functional digraph G_p such that $p[0]$ is undefined and $p[1] = 0$

```
1. Compute min(T_v) and level(v) for each v ∈ T
2. Create P corresponding to the path 1 ⇝ 0
3. Compute μ_{⇑⇓}(v) for each v ∈ P excluding 1 and 0
4. Identify all nodes f_1, f_2, ..., f_k such that μ_{⇑⇓}(f_i) = f_i
5. f_0 = 1; f_{k+1} = 0
6. if ⇑⇓ = down then
7.    for i = 1 to k in parallel do f_i = pred[f_{i+1}]
8. for i = 1 to k in parallel do p[f_i] = p[f_{i-1}]
9. p[1] = 0
10. if INVERTEDGES then
11.    for v ∈ P in parallel do p[p[v]] = v
```

Initially we compute, for each node v, $\min(T_v)$: the minimum value in the subtree rooted at 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 Rake technique. With the same bounds we can compute the top-down level of each node (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 $\mu_{\Uparrow\!\downarrow}(v)$ 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 $\Uparrow\!\downarrow$ = down then T_P have to be rooted in 0, otherwise it must be rooted in 1. To order the f_i values we can use Prefix-Sum on vector P assigning 1 to nodes v such that $\mu_{\Uparrow\!\downarrow}(v) = v$ and 0 otherwise.

The three cycles of lines 7, 8, and 11 require O(1) 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 Brent's Theorem all these operations can be scheduled on $O(n/\log n)$ processors in $O(\log n)$ time. The overall cost is linear and thus the algorithm is optimal.

Parallel Decoding. The most demanding step in the parallel decoding algorithm is the computation of characteristic nodes. It can be obtained in $O(\log n)$ time with $O(n)$ processors on a CREW PRAM in a Pointer Jumping like fashion: for each node we follow the outgoing edge searching for the max/min value in the ascending path (the parameter μ discriminate 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 detailed in Program 5: the value $\texttt{asc}(v)$ is the min/max value in the ascending path of v. At each step $\texttt{asc}(v)$ is compared with $\texttt{asc}(p[v])$ and eventually updated. Notice that we explicitly flag whether the value $\texttt{asc}(v)$ comes from v itself or has been encountered in the proper ascending path. At the end of the $\log n$ iterations if $\text{asc}(v) = v$ and $\text{self}(v)$ is false, we can state that v is the min/max

Program 5. Parallel Identification of Characteristic Nodes

```
1. p[0] = 0
2. for each node v ∈ T in parallel do asc(v) = v; self(v) = true
3. for j = 1 to ⌈log n⌉ do
4.    for each node v ∈ T in parallel do
5.       if asc(p[v]) = μ{asc(p[v]), asc(v)} then
6.          asc(v) = asc(p[v]); self(v) = false
7.       p[v] = p[p[v]]
```

Program 6. Dandelion-Like Parallel Decoding Algorithm

Parameters: μ, $\Downarrow$, INVERTEDGES
Input: a functional digraph G_p such that $p[0]$ is undefined and $p[1] = 0$
Output: a tree T represented by its parent vector p

```
1. Identify all characteristic nodes f_1, f_2, ..., f_k according with μ and ⇓
2. if INVERTEDGES then
3.    for each v ∈ cycles in parallel do p[p[v]] = v
4. f_0 = 1; f_{k+1} = 0
5. if ⇓ = down then
6.    for i = 1 to k in parallel do f_i = pred(f_i)
7. for i = 0 to k − 1 in parallel do
8. p[f_i] = p[f_{i+1}]
9. p[f_k] = 0
```

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 characteristic nodes $f_1, f_2, \ldots, f_k$. We assume that a copy of p is used in Program 5, since the original vector p will be required latter in the decoding.

Once characteristic nodes have been identified, a further Pointer Jumping can be used to broadcast a flag in their ascending paths, thus identifying all nodes belonging to some cycle. The rest of the Decoding Algorithm proceeds as detailed in Program 6. Namely, all the three cycles of lines 3, 6, and 8 require O(1) time with n processors, provided that the *pred* values are computed for all nodes belonging to any cycle (as described in Parallel Encoding for nodes in P).

6 Conclusions and Open Problems

Concluding, we want to remark that also for the Blob code [11,14] it is possible to obtain parallel algorithms with the very same costs of those presented in this paper. This result can be obtained combining ideas presented in this paper with the redefinition of the Blob code in terms of transformation of trees in to functional digraphs given in [7]. With respect to Table 1, it remains an open problem to study parallel algorithms for the Chen code.

References

1. Boppana, V., Hartanto, I., Fuchs, W.: Full Fault Dictionary Storage Based on Labeled Tree Encoding. In: IEEE VTS 1996, pp. 174–179 (1996)
2. Garg, V., Agarwal, A.: Distributed Maintenance of a Spanning Tree Using Labeled Tree Encoding. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 606–616. Springer, Heidelberg (2005)
3. Deo, N., Kumar, N., Kumar, V.: Parallel Generation of Random Trees and Connected Graphs. Congr. Num. 130, 7–18 (1998)
4. Reeves, C., Rowe, J.: Genetic Algorithms: A Guide to GA Theory. Springer, Heidelberg (2003)
5. Cayley, A.: A Theorem on Trees. Quart. J. Math. 23, 376–378 (1889)
6. Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. Archiv der Mathematik und Physik 27, 142–144 (1918)
7. Caminiti, S., Petreschi, R.: String Coding of Trees with Locality and Heritability. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 251–262. Springer, Heidelberg (2005)
8. Chen, W.: A General Bijective Algorithm for Increasing Trees. Syst. Sci. Math. Sci. 12, 194–203 (1999)
9. Deo, N., Micikevičius, P.: A New Encoding for Labeled Trees Employing a Stack and a Queue. Bulletin of ICA 34, 77–85 (2002)
10. Eğecioğlu, Ö., Remmel, J.: Bijections for Cayley Trees, Spanning Trees, and Their q-Analogues. J. Comb. Th. 42A, 15–30 (1986)
11. Kreweras, G., Moszkowski, P.: Tree codes that preserve increases and degree sequences. J. Disc. Math. 87, 291–296 (1991)
12. Moon, J.: Counting Labeled Trees. William Clowes and Sons, London (1970)
13. Neville, E.: The Codifying of Tree-Structure. In: Proc. of Cambridge Philosophical Soc., vol. 49, pp. 381–385 (1953)
14. Picciotto, S.: How to Encode a Tree. PhD thesis, U. California, San Diego (1999)
15. Caminiti, S., Finocchi, I., Petreschi, R.: On Coding Labeled Trees. TCS 382, 97–108 (2007)
16. Caminiti, S., Deo, N., Micikevičius, P.: Linear-time Algorithms for Encoding Trees as Sequences of Node Labels. Congr. Num. 183, 65–75 (2006)
17. Chen, H., Wang, Y.: An Efficient Algorithm for Generating Prüfer Codes from Labelled Trees. TCS 33, 97–105 (2000)
18. Deo, N., Micikevičius, P.: 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)
19. Greenlaw, R., Halldórsson, M., Petreschi, R.: On Computing Prüfer Codes and Their Corresponding Trees Optimally in Parallel. In: JIM 2000 (2000)
20. Caminiti, S.: On Coding Labeled Trees. PhD thesis, Sapienza U. of Rome (2007)
21. Paulden, T., Smith, D.: Recent advances in the study of the dandelion code, happy code, and blob code spanning tree representations. In: IEEE CEC 2006, pp. 2111–2118 (2006)
22. Gottlieb, J., Julstrom, B., Raidl, G., Rothlauf, F.: Prüfer Numbers: A Poor Representation of Spanning Trees for Evolutionary Search. In: GECCO 2001, pp. 343–350 (2001)
23. Wen, X., Vishkin, U.: PRAM-on-Chip: First Commitment to Silicon. In: ACM SPAA 2007, pp. 301–302 (2007)
24. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)