Linear-time Algorithms for Encoding Trees as Sequences of Node Labels

Paulius Micikevičius * Saverio Caminiti †
Narsingh Deo ‡

Abstract

In this paper we present $O(n)$-time algorithms for encoding/decoding n-node labeled trees as sequences of $n-2$ node labels. All known encodings of this type are covered, including Prüfer-like codes and the three codes proposed by Picciotto - the happy, blob, and dandelion codes. The algorithms for Picciotto's codes are of special significance as previous publications describe suboptimal approaches requiring $O(n \log n)$ or even $O(n^2)$ time.

Keywords: Labeled Tree, Code, Prüfer, Picciotto.

1 Introduction

A tree code is any bijection between the set of all labeled unrooted trees on n nodes and $n-2$ tuples of node labels. The first code was proposed by Prüfer [21] in 1918 in order to prove Cayley's theorem that there are n^{n-2} distinct labeled trees on n nodes. A number of different bijections have been proposed since then, including the ones by Glicksman, Neville, Eğecioğlu, Picciotto, Deo and Micikevičius. These codes can be classified into two categories based on their encoding approach. Deletion (also known as Prüfer-like) codes iteratively delete leaf nodes, recording their neighbors to the code. Transformation codes convert trees into directed graphs and then record the nodes' parents to the code.

*Department of Computer Science, Armstrong Atlantic State University, 11935 Abercorn Street, Savannah, GA 31419-1997, USA. E-mail: paulius@cs.armstrong.edu.

†Dipartimento di Informatica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. E-mail: caminiti@di.uniroma1.it.

‡School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816-2362, USA. E-mail: deo@cs.ucf.edu.

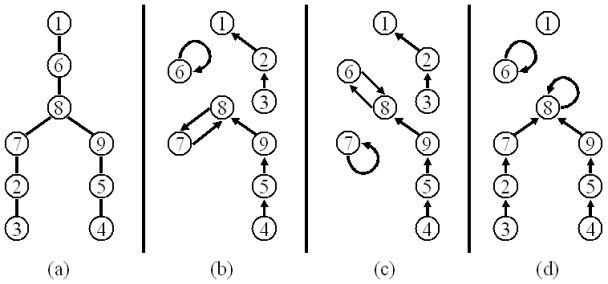

Figure 1: a) An 9-node unrooted tree. b) Digraph after Dandelion encoding. c) Digraph after Happy encoding. d) Digraph after Blob encoding.

Various tree codes have different structural properties and have found applications in graph theory and computer science. For example, Agnarsson et al [1] used the queue-based code to derive the order of the expected radius/diameter of a random labeled tree. Many graph-theoretic theorems for trees have simple proofs when using tree codes [15]. Computer science applications include efficient generation of random trees and genetic algorithms for optimization problems. The latter is of special interest and has generated a number of publications on tree-code use as well as genetic properties of various codes [4, 7, 10, 12, 13, 19, 22]. Since genetic algorithms require frequent decoding of the chromosomes (tree codes), time-optimal algorithms are critical in reducing execution time. While all known tree codes admit $O(n)$ algorithms, genetic algorithm literature describes suboptimal $O(n \log n)$ or even $O(n^2)$ algorithms for transformation codes. In this paper we present linear-time algorithms for encoding and decoding both deletion and transformation codes.

1.1 Notation and Assumptions

We assume that a tree T is stored in the adjacency-lists data structure, which allows checking and updating a node's degree in $O(1)$ time. The code is stored in array C. The nodes are labeled with integers 1 through n. Degree of node v is denoted by $deg(v)$. For a rooted tree: the parent of node v is denoted by $p(v)$, $max(v)$ refers to the greatest node on a directed path from v to the root, $\mu(v)$ denotes the greatest node on a directed path from v, where the length of the path is at least 1.

2 Deletion Codes

All deletion codes iteratively delete leaf nodes of a tree until only one edge remains. Each time a node is deleted, its only neighbor is appended to the code. Prüfer's code [21] deletes the leaf with the smallest label each time. In 1953 Neville [16] described three codes, the first one of which is identical to Prüfer's approach. Neville's second code deletes the leaves in ascending order of the labels level-by-level: first, all the original leaves of the tree are deleted, next the leaves of the remaining subtree are deleted, *etc.* Neville's third code at each step deletes the leaf node with the smallest label, except that a newly created leaf is always deleted next (no matter what its label is). In 2002 Deo and Micikevičius [6] proposed a code where leaf nodes are kept in a queue and the node at the head is always deleted next. The initial queue contains leaves of the original tree arranged in ascending order of their labels, newly created leaves are appended to the tail. A preliminary classification of deletion codes, including a brand new stack-based code, appears in [5, 15]. Surveys by Caminiti et al [2, 3] formulate these codes in terms of sorting pairs of integers.

2.1 Encoding Algorithms

Deletion codes are differentiated by the order in which they delete leaf nodes. To formalize the process, we assume that at any point the leaves are stored in a list. If nodes are deleted level-by-level (as is done by Neville's second code) a separate list is maintained for each level. Thus, deletion codes are classified by two general characteristics: list type and the number of lists. We consider three list types: stack (first-in, first-out), queue (first-in, last-out), and sorted list. There are two choices for the number of lists: single and multiple. Prior to encoding the list contains the leaves arranged in ascending order of the labels. For simplicity of presentation, we list generic algorithms for single and multiple-list encoding separately.

Algorithm: Generic Single-list Encoding Algorithm

```
1: L ← leaves of T
2: for i ← 1 to n − 2 do
3:     v ← node removed from the head of L
4:     C[i] ← neighbor of v
5:     delete v from T
6:     if deg(C[i]) = 1 then add C[i] to L
```

The initial list on line 1 can be created in $O(n)$ time by examining each node's adjacency-list. Operations on lines 3 through 5 all take $O(1)$ time, regardless of the list type. Line 6 takes $O(1)$ time if L is a stack or a queue, leading to $O(n)$ complexity for the entire algorithm. Multiple-list encoding

performs the same operations, except that it adds newly created leaves to the "next" list. As soon as the current list is exhausted, the "next" list becomes current (line 8) and the process is repeated. This ensures that nodes are deleted level-by-level.

Algorithm: Generic Multiple-list Encoding Algorithm

```
1: k ← 0
2: L[k] ← leaves of T
3: for i ← 1 to n − 2 do
4:    v ← node removed from the head of L[k]
5:    C[i] ← neighbor of v
6:    delete v from T
7:    if deg(C[i]) = 1 then add C[i] to L[k + 1]
8:    if L[k] = ∅ then k ← k + 1
```

Straightforward implementation where the lists are either stacks or queues takes $O(n)$ time. Details of $O(n)$ implementations for sorted lists are discussed in Section 2.3. As was proved in [5], single and multiple-list codes are identical if lists are queues. Thus, there are 5 distinct encodings. The deletion codes for the tree in Figure 1a are:

- Single-list sorted (Prüfer's): $(6, 2, 7, 5, 9, 8, 8)$;
- Multiple-list sorted (Neville's second): $(6, 2, 5, 7, 9, 8, 8)$;
- Single-list stack (Neville's third): $(6, 8, 2, 7, 8, 5, 9)$;
- Multiple-list stack (proposed in [5]): $(6, 2, 5, 9, 7, 8, 8)$;
- Queue (D-M code proposed in [6]): $(6, 2, 5, 8, 7, 9, 8)$.

2.2 Decoding Algorithms

Generic decoding algorithms reverse the encoding process to construct a tree from its code. Initially, the list contains nodes missing from the code. A node is added to the list (lines 5 and 6) if it does not appear in the code to the right of the current position i. This test can be carried out in $O(1)$ time assuming some preprocessing. First, all nodes are marked as unvisited. Next, the code is scanned from right to left. Each time an unmarked node is encountered it is marked as visited and the position is marked as the rightmost occurrence. Note that the unmarked nodes are the ones added to the initial list on line 1. Preprocessing traverses the code once, requiring $O(n)$ time.

Algorithm: Generic Single-list Decoding Algorithm

```
1: L ← nodes that do not appear in code C
2: for i ← 1 to n − 2 do
3:    v ← node removed from the head of L
4:    add edge {v, C[i]} to T
5:    if i is the rightmost position of v in C then
6:       add v to L
7: v ← node removed from the head of L
8: add edge {v, C[n − 2]} to T
```

The same type of list L must be used by corresponding encoding and decoding algorithms. Multiple-list decoding algorithm performs the same operations, except that it maintains multiple lists. Straightforward implementations for stacks and queues take $O(n)$ time since all operations inside the loop take constant time and preprocessing requires $O(n)$ time.

Algorithm: Generic Multiple-list Decoding Algorithm

```
1: k ← 0
2: L[k] ← nodes that do not appear in code C
3: for i ← 1 to n − 2 do
4:    v ← node removed from the head of L[k]
5:    add edge {v, C[i]} to T
6:    if i is the rightmost position of v in C then
7:       add v to L[k + 1]
8:    if L[k] = ∅ then k ← k + 1
9: v ← node removed from the head of L
10: add edge {v, C[n − 2]} to T
```

2.3 $O(n)$ Encoding/Decoding with Sorted Lists

Straightforward implementations of the above algorithms with sorted lists require inserting a newly created leaf into a correct sorted position, leading to $O(n \log n)$ time.

Single-list. To the best of our knowledge, the first $O(n)$-time approach for single sorted list (Prüfer) code was described by Kilingberg [17](page 271). The method was subsequently rediscovered in [8]. The idea is to examine the nodes in ascending order of the labels. Each time a leaf is encountered, its neighbor is recorded to the code and it is deleted from the tree. If a newly created leaf is smaller than the current node, the leaf is processed next (while-loop on line 6, otherwise the next node in order of the labels is examined (line 10). This method requires $O(n)$ time since each node is examined at most twice: when it is visited by the traversal and when it becomes a leaf. Pseudo-code below uses variables i and v to indicate the current code position and the current node being traversed, respectively.

Algorithm: Single Sorted List Encoding

```
1: i ← 1; v ← 1
2: while i ≤ n − 2 do
3:    if deg(v) = 1 then
4:       C[i] ← neighbor of v
5:       delete v from T
6:       while deg(C[i]) = 1 and C[i] < v and i < n − 2 do
7:          C[i + 1] ← neighbor of C[i]
8:          delete C[i] from T
9:          i ← i + 1
10:      i ← i + 1
11:   v ← v + 1
```

The decoding process simply reverses the encoding. The nodes are traversed in ascending order of the labels. If a node that does not appear in the code is encountered, an edge is added between that node and the one in the current position of the code i. If i is the rightmost occurrence of node $C[i]$ in the code, it is marked as missing from the code and, if its label is smaller than the one current in the traversal, it is processed next. The rightmost-position test requires $O(1)$ time after preprocessing as described for the generic algorithm above.

Algorithm: SINGLE SORTED LIST DECODING

```
1: mark all nodes that do not appear in code C
2: i ← 1; v ← 1
3: while i ≤ n − 2 do
4:    if v is marked then
5:       add edge {C[i], v} to T
6:       if i is the rightmost position for C[i] in C then mark C[i]
7:       while C[i] is marked and C[i] < v and i < n − 2 do
8:          add edge {C[i], C[i + 1]} to T
9:          i ← i + 1
10:         if i is the rightmost position for C[i] in C then mark C[i]
11:      i ← i + 1
12:   v ← v + 1
13: add edge {C[n − 2], n} to T
```

Multiple-list. The leaves are deleted level-by-level. Let the leaves of the original tree be level-0 nodes. The nodes that are leaves after all level-0 nodes are deleted are level-1 nodes, *etc.* Alternatively, if a tree is rooted at its center, the level of a node is defined recursively: $l(v)$=0 if v is a leaf; $l(v) = max\{l(u)$, where u is a child of $v\}$ otherwise. A center of a tree can be found in $O(n)$ time with two Depth-First Traversals (DFT). The tree can be rooted and levels for all nodes can be computed in $O(n)$ time with another DFT using the recursive definition. Next, for each node v create a pair $(l(v), v)$ and sort these pairs in ascending lexicographic order using radix sort ($O(n)$ time). The sorted pairs specify precisely the order in which the nodes are deleted from the tree during encoding. Thus, $C[i]$ is the parent (the tree has been rooted) of the node corresponding to the i^{th} sorted pair.

During decoding, we reconstruct the same sorted list of pairs. First, we mark all the rightmost occurrences of nodes in the code (as described earlier, this takes $O(n)$ time). The nodes missing from the code are assigned level 0. The levels of the nodes that do appear in the code are computed in $O(n)$ time as follows. Let $numl(k)$ be the number of nodes at level k. Level-1 nodes appear in the marked positions among the first $numl(0)$ positions of the code (since during encoding a node becomes a leaf exactly when it is written to the code for the last time). Level-2 nodes appear in the marked positions among the next $numl(1)$ positions of the code, *etc.*

In general, level-k nodes appear in marked positions of the code between positions $\sum_{i=0}^{k-2} numl(i)$ and $\sum_{i=0}^{k-1} numl(i)$. Lexicographic order of the pairs is obtained in $O(n)$ time. Edges between $C[i]$ and the node corresponding to the i^{th} sorted pair are added to the tree. The tree is completed by adding the edge between $C[n-2]$ and the node corresponding to the $n-1^{st}$ pair.

3 Transformation Codes

Transformation codes convert a given tree into a digraph before recording the nodes' parents to the code. First, the tree is rooted at node 1 and all edges are assigned direction from child to parent. Next, the digraph is transformed, possibly creating cycles, to satisfy a specific property. Finally, the parents of $n-2$ nodes are recorded to the code. It is the transformation process and the property that differentiate various codes.

In her PhD thesis [20] Picciotto described three codes: Happy, Blob, and Dandelion. As she points out, these codes are related to previous results. One gives explicitly a bijection that is implicit in the Orlin's proof of Cayley's theorem [18]. Another is based on a proof of Knuth [14]. The last one is an implementation of the Joyal's pseudo-bijective proof of Cayley's theorem [11] and is equivalent to the one introduced by Eğecioğlu and Remmel [9]. The three transformation codes for the tree in Figure 1a are:

- Dandelion: $(2, 5, 9, 6, 8, 7, 8)$;
- Happy: $(7, 2, 5, 9, 6, 8, 8)$;
- Blob: $(2, 5, 9, 8, 7, 6, 8)$.

3.1 Preprocessing

The transformation codes operate on trees rooted at node 1. An unrooted tree can be rooted (node-parents computed) in $O(n)$ time with a Depth-First Traversal (DFT).

For each node v, the Dandelion and Blob encoding algorithms require the knowledge of $max(v)$ – the greatest node on the directed path from v to the root (including v). A DFT can compute these values during the rooting process.

All transformation decoding algorithms require $\mu(v)$ for each node v. $\mu(v)$ is the greatest node on the directed path from $p(v)$. Note that $\mu(v) = v$ implies that v is the greatest node in a cycle. Again, a DFT algorithm is easily modified to compute μ values in $O(n)$ time (see [4] for details).

3.2 Dandelion code

The Dandelion code transforms a tree into a digraph rooted at node 1 that contains the edge $(2, 1)$. This is accomplished by iteratively swapping the parent of node 2 with $max(p(2))$ - the node with the greatest label on a path from $p(2)$ to the root. Note that each swap creates a cycle and $max(p(2))$ is the greatest label in that cycle. All operations inside the loops take constant time, leading to $O(n)$ time for the encoding algorithm.

Algorithm: DANDELION ENCODING

```
1: v ← p(2)
2: while v ≠ 1 do
3:     swap v and p(max(v))
4: for i ← 1 to n − 2 do
5:     C[i] ← p(i + 2)
```

To reconstruct a tree from its code, we first add edges $(2, 1)$ and $(i + 2, C[i])$, where $1 \leq i \leq n - 2$, to an empty digraph. Next, for each cycle in the digraph we identify the nodes with the greatest label. Note that every such node $v = \mu(v)$. The parents of these nodes were swapped during encoding (line 3), thus we can reconstruct the original tree.

Algorithm: DANDELION DECODING

```
1: p(2) ← 1
2: for i ← 1 to n − 2 do
3:     p(i + 2) ← C[i]
4: for v = 3 to n do
5:     if μ(v) = v then
6:         swap p(2) and p(v)
```

Since all operations inside the loops take $O(1)$ time after preprocessing, the entire algorithm requires $O(n)$ time. Another $O(n)$ approach for endocing/decoding Dandelion code was proposed by Paulden and Smith [19].

3.3 Happy code

The Happy code traverses the path from 2 to 1, placing the intermediate nodes in cycles. The first cycle is started with $p(2)$. Every time a node whose parent is greater than the cycle starter node is reached, a new cycle containing that one node is started (lines 5 through 7). Other nodes are inserted into the current cycle immediately after the starter node (lines 9 and 10). Note that the starter node always has the greatest label on its cycle. Since the algorithm traverses the path from 2 to 1 once, the algorithm requires $O(n)$ time.

Algorithm: HAPPY ENCODING

```
1: starter ← 0
2: while p(2) ≠ 1 do
3:     v ← p(2)
4:     p(2) ← p(v)
5:     if v > starter then
6:         starter ← v
7:         p(starter) ← starter
8:     else
9:         p(v) ← p(starter)
10:        p(starter) ← v
11: for i ← 1 to n − 2 do
12:     C[i] ← p(i + 2)
```

To reconstruct a tree from its code, we first add edges $(2, 1)$ and $(i+2, C[i])$, where $1 \leq i \leq n-2$, to an empty digraph. Next, the decoding algorithm considers each cycle-starter node (a node equal to its μ-value) in descending order. Each starter node's cycle is traversed (lines 6 through 11), reinstating the edges originally in the rooted T.

Algorithm: HAPPY DECODING

```
1: p(2) ← 1
2: for i ← 1 to n − 2 do
3:     p(i + 2) ← C[i]
4: for starter ← n downto 3 do
5:     if μ(starter) = starter then
6:         v = p(starter)
7:         while p(2) ≠ starter do
8:             next ← p(v)
9:             p(v) ← p(2)
10:            p(2) ← v
11:            v ← next
```

The algorithm requires $O(n)$ time as it visits each node at most twice: during the traversal by the for-loop on line 4 and while traversing a cycle by the while-loop.

3.4 Blob code

Algorithm: BLOB ENCODING

```
1: last ← p(n)
2: for v ← n − 1 downto 2 do
3:     if max(v) = v then
4:         swap p(v) and last
5: for i ← 1 to n − 2 do
6:     C[i] ← p(i + 1)
```

Picciotto's description of the Blob code adds nodes to the *macro-node* in decreasing order of the labels, possibly modifying their parents. We forgo

the use of the macro-node and simply describe the encoding in terms of parent modification. The parent of node v is changed to the *last* node if v is the greatest label on the path from v to the root ($v = max(v)$). Since all operations in the loops take $O(1)$ time, the algorithm requires $O(n)$ time.

Algorithm: BLOB DECODING
1: $p(n) \leftarrow 1$
2: **for** $i \leftarrow 1$ **to** $n-2$ **do**
3: $\quad p(i+1) \leftarrow C[i]$
4: $last = 1$
5: **for** $v = 2$ **to** n **do**
6: $\quad$ **if** $\mu(v) \leq v$ **then**
7: $\quad\quad$ `swap` $p(v)$ `and` $last$

Even though encoding does not explicitly reassign $p(n)$, decoding algorithm assumes that $p(n) = 1$. Thus, decoding is started by adding edges $(n, 1)$ and $(i + h1, C[i])$, where $1 \leq i \leq n-2$, to an empty digraph. Note that the path from each v, such that $max(v) > v$, to $max(v)$ is the same in the original tree and the initial digraph. Furthermore, there is no path from $v = max(v)$ to a node greater than v in the digraph. Thus, utilizing $\mu(v)$ we can reconstruct the parent swaps in $O(n)$ time. Proof of correctness appears in [4].

4 Conclusions

We have described $O(n)$-time algorithms for both computing and decoding all known deletion and transformation tree codes. To the best of our knowledge, no $O(n)$ algorithms have been published for the Blob and Happy codes, even though they are used by the genetic algorithm community. The deletion algorithms assume non-rooted trees. However, these algorithms can be easily extended to operate on rooted trees by ensuring that the chosen root is never added to the list of leaves (and, therefore, is never deleted). This extension would not affect the time-complexity as it would simply add an if-statement.

References

[1] G. Agnarsson, N. Deo, and P. Micikevičius. On the Expected Number of Level-i Nodes in a Random Labeled Tree. *Bulletin of the Institute of Combinatorics and its Applications (ICA)*, 41:51–60, 2004.

[2] S. Caminiti, I. Finocchi, and R. Petreschi. A Unified Approach to Coding Labeled Trees. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN '04), LNCS 2976*, pages 339–348, 2004.

[3] S. Caminiti, I. Finocchi, and R. Petreschi. On Coding Labeled Trees. *To appear on Theoretical Computer Science*, 2006.

[4] S. Caminiti and R. Petreschi. String Coding of Trees with Locality and Heritability. In *Proceedings of the 11th International Conference on Computing and Combinatorics (COCOON '05), LNCS 3595*, pages 251–262, 2005.

[5] N. Deo and P. Micikevičius. Prüfer-like Codes for Labeled Trees. *Congressus Numerantium*, 151:65–73, 2001.

[6] 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*, 34:77–85, 2002.

[7] 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*, page 754, 2000.

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

[9] 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.

[10] 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*, pages 343–350, 2001.

[11] A. Joyal. Une Theorie Combinatoire des Series Formelles. *Advances in Mathematics*, 42:1–81, 1981.

[12] B.A. Julstrom. The Blob Code: A Better String Coding of Spanning Trees for Evolutionary Search. In *Proceedings of Representations and Operators for Network Problems*, pages 256–261, 2001.

[13] 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*, pages 585–590, 2005.

[14] D.E. Knuth. Oriented Subtrees of an Arc Digraph. *Journal of Combinatorial Theory*, 3:309–314, 1967.

[15] P. Micikevičius. *Parallel graph algorithms for molecular conformation and tree codes*. PhD thesis, University of Central Florida, 2002.

[16] E.H. Neville. The Codifying of Tree-Structure. In *Proceedings of Cambridge Philosophical Society*, volume 49, pages 381–385, 1953.

[17] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms*. Academic Press, New York, 1978.

[18] J.B. Orlin. Line-Digraphs, Arborescences, and Theorems of Tutte and Knuth. *Journal of Combinatorial Theory*, 25:187–198, 1978.

[19] 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):108–122, 2006.

[20] S. Picciotto. *How to Encode a Tree*. PhD thesis, University of California, San Diego, 1999.

[21] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:142–144, 1918.

[22] G. Zhou and M. Gen. A note on genetic algorithms for degree-constrained spanning tree problems. *Networks*, 30:91–95, 1997.