A New Encoding for Labeled Trees Employing a Stack and a Queue

Narsingh Deo and Paulius Micikevicius
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362
Email: deo@cs.ucf.edu, pmicikev@cs.ucf.edu

Abstract

A novel algorithm for encoding finite labeled trees is proposed in this paper. The algorithm establishes a one-to-one mapping between trees of order n and (n – 2)-tuples of the node labels. This encoding is similar to those proposed by Prüfer [8] and Neville [6], except that our encoding scheme has additional properties (not present in earlier codes), which allows us to efficiently determine the center(s), radius, and diameter directly from the code, without having to explicitly construct the tree. We also present conditions for modifying the proposed code in order to obtain trees at distance one from the original. A queue is used during encoding while a stack is used during decoding. Both encoding and decoding require O(n) time.

Introduction

Labeled trees are of interest in several practical and theoretical areas of computer science. For example, some networks, such as Ethernet, are required to have one and only one path between every terminal device, and are therefore trees. Moreover, labeling the nodes is necessary since each device in such network is a distinct entity. Many algorithms for analysis of general networks require their spanning trees.

Encoding trees into (n – 2)-tuples of node labels is employed in genetic algorithms, where all possible solutions are represented as strings of integers. Genetic algorithms have been used in heuristics for computing constrained minimum spanning trees – minimum-weight spanning trees satisfying an additional constraint, such as the number of leaves, maximum degree, or diameter of the tree [1, 9].

Existing codes for labeled trees

Prüfer's method [5, 8] encodes a tree by iteratively deleting the leaf node with the smallest label and recording its neighbor, until only one edge remains. For example, the Prüfer code for the tree in Figure 1 is (9, 7, 7, 3, 10, 4, 4, 7, 1). An O(n)-time algorithm for constructing a tree from its Prüfer code has been proposed by Kilingsberg [7, p. 271]. Kilingsberg's method can also be adopted to obtain the Prüfer code for a given tree of order n in O(n) time. One drawback

of the Prüfer code is its lack of structure for directly determining the diameter or the center(s) from the code without first constructing the tree.

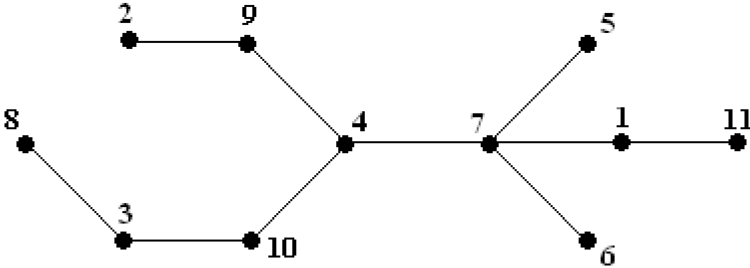

Figure 1 A tree

Neville proposed three methods for encoding trees in [6]. In all three algorithms an arbitrary node is selected as the root node and is never deleted from the tree. Moon generalized Neville's algorithms by not requiring a root node [7]. The first encoding proposed by Neville is equivalent to the Prüfer's method when the node with the largest label is chosen as the root, thus the code for the tree in Figure1 is (9, 7, 7, 3, 10, 4, 4, 7, 1).

Neville's second method proceeds r in stages, where r is the radius of the tree. In each stage the leaves of the current subtree are deleted in ascending order of the labels, and their neighbors are recorded. The process is repeated until only one edge remains (thus, if the radius of a tree is even, all but one leaf will be removed in the last stage). For example, the code for the tree in Figure 1 is (9, 7, 7, 3, 1, 7, 10, 4, 4). This encoding leads to a simple algorithm for computing the diameter of a tree directly from the code. However, there are no known $O(n)$-time algorithms for encoding or decoding trees using Neville's second method since leaf nodes must be sorted in each stage of the algorithm.

The third encoding due to Neville starts by deleting the leaf node with the smallest label and recording its neighbor, v. If v is a leaf in the resulting subtree, v is deleted next, otherwise a leaf with the smallest label is deleted. The process is repeated until one edge remains. With this method the code for the tree in Figure 1 is (9, 4, 7, 7, 3, 10, 4, 7, 1).

Glicksman exploited the bipartite property of trees to show a one-to-one mapping from a pair of label-tuples to trees [2]. While the tuples contain $(n – 2)$ labels between them, Glicksman did not give a method for partitioning a $(n – 2)$-tuple into two. To the best of our knowledge, there is no known method for encoding trees into $(n – 2)$-tuples using Glicksman's approach.

Notation and data structures

T: a tree, $T = (V(T), E(T))$
$E(T)$: the set of edges in T
$V(T)$: the set of nodes of T
n: number of nodes in T
$deg(v)$: degree of node v
$A[k]$: the kth element in some linear array A

It is assumed that adjacency lists are used to store a tree, which allows the degrees of all nodes to be computed in $O(n)$ time and a neighbor of a leaf node to be found in constant time. The nodes of a tree are labeled 1, …, n. Array indices start with 1. Pseudo-code notation is similar to that described in [3].

Encoding a labeled tree

The proposed algorithm may be viewed as a modification of Neville's second method. The leaves of the given tree are deleted in ascending order of their labels, while the leaves of subsequent subtrees are deleted in the order in which they become leaves. When a node is deleted, its neighbor is appended to the code. The leaves of the current subtree are stored in a queue Q. The code for the tree in Figure 1 is (9, 7, 7, 3, 1, 4, 10, 7, 4). A pseudo-code description is given in Figure 2.

Input: T, a labeled tree of order n
Output: C, the code for T

```
1.    for i ← 1 to n do
2.        if deg(i) = 1 then
3.            add i to the tail of Q

4.    for i ← 1 to (n – 2) do
5.        remove u from the head of Q
6.        C[i] ← v, where v is the node adjacent to u
7.        remove u from T
8.        if deg(v) = 1 then
9.            add v to the tail of Q
```

Figure 2 Algorithm for encoding a tree

The loop on lines 1 through 3 adds the leaves of the given tree to the queue Q in ascending order of their labels. The loop on lines 4 through 9 records the code of the tree. The node to be deleted next is selected on line 5. Node v, adjacent to the selected node u, is appended to the tree code on line 6. After

node u is deleted from the tree on line 7 the degree of v is checked on line 8. If v has become a leaf, it is added to the tail of the queue.

It is assumed that all node degrees are computed and stored in an array when the nodes are examined in the loop on lines 1-3. This requires O(n) time, since each edge has to be examined twice when adjacency list data structure is used to store a tree. Each call to *deg*(v) function can therefore execute in O(1) time. Since only degree 1 nodes are removed on line 7, the degree of one other node has to be updated each time. Thus, removing a node requires O(1) time.

Theorem 1. Tree encoding requires O(n) time.

Proof. The loop on lines 1 through 3 requires O(n) time. Queue operations on lines 5 and 9 can be implemented in constant time. Finding the neighbor of node u on line 6 requires constant time since the degree of node u is one. As discussed in the previous paragraph, operations on lines 7 and 8 take O(1) time each. Thus, since the loop on lines 4 through 9 performs O(n) iterations and each iteration requires constant time, a tree is encoded in O(n) time. ■

Constructing a tree from the code

The algorithm for constructing a tree from its code starts at the last position of the given code and traverses it from right to left, pushing each label encountered for the first time onto the top of the stack. Next, all the labels missing from the code are pushed onto the stack in descending order. Edges are added by joining the kth node in the code with the kth node popped off the stack, where $k \in [1, n - 2]$. The last edge is formed by joining the last node in the code with the $(n - 1)^{st}$ node popped off the stack. A pseudo-code description is given in Figure 3.

A temporary array *used* indicates which nodes have been encountered while traversing the code. If *used*[v] = 1 node v has been encountered, if *used*[v] = 0 it has not. The two loops on lines 3 through 9 push n node labels onto the top of the stack S: the first loop pushes the internal nodes onto S in reverse order of their last appearance in the code (*i.e.* the node whose last appearance in the code is rightmost will be pushed on first), while the second loop pushes the leaves (the nodes missing from the code) onto the stack in descending order of their labels. $(n - 2)$ edges are formed by the loop on lines 10 through 12. The last edge is added on line 14.

Theorem 2. Algorithm in Figure 3 constructs the tree given its code.

Proof. Since the loop on lines 11 through 13 joins the kth node popped off the stack with the kth node in the code, it is necessary to show that the sequence in which the labels are popped off the stack is the same sequence in which the nodes were added to the queue during encoding of the tree. One also needs to show that the edge added on line 14 is the same edge that remained after encoding.

Input: C, a code of length $(n-2)$
Output: $E(T)$, the edge set of the encoded tree

```
1.   for i ← 1 to n do
2.       used[i] ← 0

3.   for i ← (n – 2) to 1 do
4.       if used[C[i]] = 0 then
5.           used[C[i]] ← 1
6.           push i onto the top of S

7.   for i ← n to 1 do
8.       if used[i] = 0 then
9.           push i onto the top of S

10.  for i ← 1 to (n – 2) do
11.      pop v off the top of S
12.      E(T) ← E(T) ∪ {(C[i], v)}
13.  pop v off the top of S
14.  E(T) ← E(T) ∪ {(C[n – 2], v)}
```

Figure 3 Algorithm for constructing a tree from its code

All the leaves of the tree (nodes missing from the code) are be popped off the stack first, in ascending order of their labels, which is the same order in which they were added to the queue during encoding.

Let's consider two internal nodes u and v, such that during encoding u was added to the queue before v. This implies that u appears in the code for the last time to the left of the last occurrence of v and therefore during decoding u is popped off the stack before v. This condition is satisfied for any two internal nodes since u and v were selected arbitrarily. Thus, the internal nodes of the tree will be popped off the stack in the same order as they were added to the queue during encoding.

Let (a, b) be the edge that remained after the tree was encoded. Since a and b were the last nodes to become leaves, they were the last two nodes added to the queue. This implies that a and b will be the last nodes popped off the stack during decoding. Without loss of generality let a be the last label in the code. Then the $(n-1)^{st}$ node popped off the stack is b. Thus, the edge added on line 14 is (a, b). ■

Theorem 3. Constructing a tree from its code requires $O(n)$ time.

Proof. All the loops execute in $O(n)$ iterations. Stack operations as well as adding an edge to a tree can be implemented in constant time. Thus, a single iteration of every loop in the algorithm requires $O(1)$ time. ■

Since every tree corresponds to a unique code, and each code corresponds to a unique tree, the two algorithms form a one-to-one mapping between the trees and their codes.

Determining the diameter of a tree from its code

The proposed code leads to an $O(n)$-time algorithm for determining the diameter and radius directly from the code, without having to construct the tree. Initially *diam*, the variable indicating the diameter, is set to zero. The algorithm proceeds in stages, during each stage incrementing *diam* by 2 and deleting all the pendant edges from the current tree. The process is repeated until all that remains is a path, at which point *diam* of the tree is incremented by the length of the path. The path may be of length zero, which will occur if a tree has one center and all the leaves are equidistant from the center.

Fundamental to our encoding scheme is that the first l nodes in the code are incident to pendant edges, where l is the number of leaves. Thus, *diam* is incremented by 2 after the first l positions in the code are traversed. As the code is traversed we also count how many of these positions are the rightmost occurrences of the corresponding labels, which gives us a new value for l (number of leaves in the remaining subtree). The process is repeated until the new value for l is two, which indicates that only a path remains.

A pseudo-code description is given in Figure 4. A number of temporary data structures are used by the algorithm. They are:

leaves: the number of leaves in the current tree.

newleaves: the number of leaves created after removing the previous leaves from the tree.

last: a linear array that marks the rightmost (last) occurrences of labels in the code. The value of *last*[i] is 1 if i is the rightmost position in the code for the node in $C[i]$ and it is 0 otherwise. For example, for the tree in Figure 1 *last* is (1, 0, 0, 1, 1, 0, 1, 1, 1).

used: a linear array indicating which nodes appear in the code. If *used*[v] = 1 node v appears in the code, if *used*[v] = 0 it does not.

current: an index variable for traversing the code. This variable is necessary since the code is traversed in segments (each corresponding to the leaves in the current subtree).

Input: C, code of length $(n - 2)$
Output: *diam*, diameter of the tree encoded by C

```
1.   diam ← 0
2.   leaves ← n
3.   for i ← 1 to n do
4.       used[i] ← 0

5.   for i ← (n – 2) to 1 do
6.       if used[C[i]] = 0 then
7.           leaves ← leaves – 1
8.           last[i] ← 1
9.           used[C[i]] ← 1
10.      else
11.          last[i] ← 0

12.  current ← 0
13.  while leaves > 2 do
14.      newleaves ← 0
15.      for i ← 1 to leaves do
16.          current ← current + 1
17.          newleaves ← newleaves + last[current]
18.      leaves ← newleaves
19.      diam ← diam + 2
20.  diam ← diam + n – current – 1
```

Figure 4 Algorithm for computing the diameter of a tree

Lines 1-4 initialize variables *diam*, *leaves*, *used*. The **for** loop on lines 5 through 11 marks the rightmost occurrences of internal nodes in the code and counts the leaf nodes. Each iteration of the **while** loop on lines 13 through 19 corresponds to an edge-deletion stage of the algorithm. The **for** loop on lines 15 through 17 logically deletes the pendant edges and counts the new leaves. During each edge-deletion stage the number of leaves is updated on line 18 and *diam* is incremented by 2 on line 19. On line 20 *diam* of the tree is incremented by the number of edges in a path that remains after the **while** loop

Algorithm 3 can also be used to determine the radius of an encoded tree, since radius is equal to $\lfloor d/2 \rfloor$. Let u be the last label in the code. Node u is the center of the tree. If a tree is bicentral, the rightmost label other than u is the other center node.

Modifying the code to obtain a tree at distance one

A tree T_1 is defined to be at distance k from a tree T_2 if $|E(T_1) \cap E(T_2)|$ = $(n-k-1)$, that is T_1 contains k edges not in T_2. Obtaining a tree at a small distance from the given tree is important in genetic algorithms, where mutation of the code should produce a solution nearby in the search space.

Let u be the label in the ith position in the code of T_1. Let v be a node whose rightmost position in the code is j, where $j > i$. If i is not the rightmost position of u, record v to the ith position in the code to obtain T_2.

Theorem 4. Changing the code as described above will result in a code for a tree at distance one from the original.

Proof. Since the codes of T_1 and T_2 differ in one position, it suffices to show that the same sequence of labels is pushed onto the stack during construction of trees T_1 and T_2 from their codes. The only nodes that can affect the sequence are u and v. Since the rightmost positions of u and v in the modified code are the same as in the given code, the sequence in which the labels are pushed onto the stack during decoding is the same for both trees. Thus, T_2 contains exactly one edge not in T_1. ■

Conclusion

In this paper we have proposed a new O(n)-time procedure for encoding a labeled tree of order n into (n – 2)-tuple of node labels. An O(n)-time algorithm for determining the diameter of the tree directly from its code has also been presented. To the best of our knowledge there is no known algorithm for computing the diameter of a tree directly from its Prüfer code; while Neville's second code does not lead to O(n)-time algorithms for encoding/decoding trees. We have also described how to modify the code of a given tree in order to obtain another tree at distance one. Determining the distance between two trees of the same order whose codes differ in exactly one position would be useful in design of genetic algorithms for constrained minimum spanning trees.

References

[1] W. Edelson, M. L. Gargano. Feasible encodings for GA solutions of constrained minimal spanning tree problems. *Proceedings of GECCO-2000*, Las Vegas, Nevada, 2000.

[2] S. Glicksman. On the representation and enumeration of trees. *Proceedings of Cambridge Philosophical Society*, vol. 59, pp. 509-517, 1963.

[3] D.L. Kreher, D.R. Stinson. Pseudocode: A LATEX Style File for Displaying Algorithms. *Bulletin of ICA*, vol. 30, pp. 11-24, 2000.

[4] V. Kumar, N. Deo, N. Kumar. Parallel generation of random trees and connected graphs. *Congressus Numerantium*, vol. 138, pp. 7-18, 1998.

[5] J. W. Moon. *Counting Labeled Trees*. William Clowes and Sons, London, 1970.

[6] E. H. Neville. The codifying of tree-structure. *Proceedings of Cambridge Philosophical Society*, vol. 49, pp. 381-385, 1953.

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

[8] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, vol. 27, pp. 142-144, 1918.

[9] G. Zhou, M. Gen. A note on genetic algorithms for degree-constrained spanning tree problems. *Networks*, vol. 30, pp. 91-95, 1997.