Theory Comput. Systems 33, 97–105 (2000)
10.1007/s002249910006

Theory of Computing Systems

An Efficient Algorithm for Generating Prüfer Codes from Labelled Trees

H.-C. Chen and Y.-L. Wang

Department of Information Management,
National Taiwan University of Science and Technology,
Taipei, Taiwan, Republic of China
ylwang@cs.ntust.edu.tw

Abstract. According to Cayley's tree formula, there are n^{n-2} labelled trees on n vertices. Prüfer gave a bijection between the set of labelled trees on n vertices and sequences of $n-2$ numbers, each in the range $1, 2, \ldots, n$. Such a number sequence is called a Prüfer code. The straightforward implementation of his bijection takes $O(n \log n)$ time. In this paper we propose an $O(n)$ time algorithm for the same problem. Our algorithm can be easily parallelized so that a Prüfer code can be generated in $O(\log n)$ time using $O(n)$ processors on the EREW PRAM computational model.

1. Introduction

Let T be a tree with n vertices. Then tree T is called a *labelled tree* if the n vertices are distinguished from one another by names such as $v_1, v_2, \ldots, v_n$. Two labelled trees are considered to be distinct if they have different vertex labels even though they might be isomorphic as graphs. For example, the two trees T_1 and T_2 in Figure 1 are labelled but T_3 is not. Moreover, T_1 and T_2 are two different labelled trees even though they are isomorphic [8].

According to Cayley's tree formula [2], there are n^{n-2} different labelled trees on n vertices. A number of proofs of Cayley's formula are known [5], [13], [19]. In [16] Prüfer used a simple one-to-one correspondence between labelled trees on n vertices and sequences of $n-2$ numbers, each in the range $1, 2, \ldots, n$. Such a number sequence

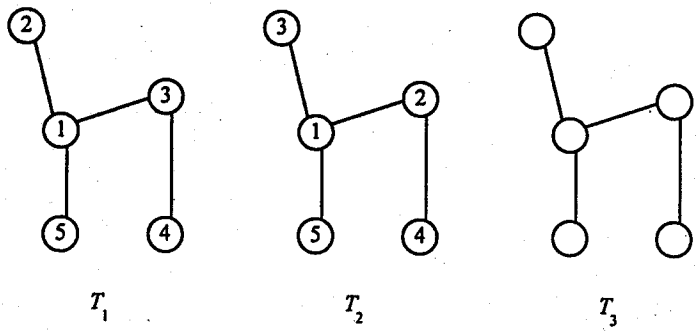

Fig. 1. Labelled and unlabelled trees.

is called a *Prüfer code*. A straightforward implementation of his proof for generating a Prüfer code takes $O(n \log n)$ time. Although the problem of producing a Prüfer code in linear time is an exercise in two books [4], [15], there exists no explicit publication of a solution. In [20] Wang et al. introduced a parallel algorithm to obtain a labelled tree from a Prüfer code. In this paper we propose an O(n) time algorithm for generating a Prüfer code from a labelled tree. Our algorithm can be easily parallelized on the EREW PRAM (Exclusive-Read-Exclusive-Write Parallel Random Access Machine) computational model, and a Prüfer code can be generated in $O(\log n)$ time with $O(n)$ processors.

The remaining part of this paper is organized as follows. In Section 2 we introduce a straightforward implementation of Prüfer's proof. In Section 3 a linear time algorithm for generating the Prüfer code of a labelled tree is proposed. The parallelization of our algorithm is shown in Section 4. Finally, in Section 5, we give the conclusion of this paper.

2. Preliminaries

In this section we discuss how to obtain the Prüfer code of a labelled tree by Prüfer's proof [7]. Before introducing the algorithm, we define some notation which will be used later.

Let T be a labelled tree whose vertices are numbered from 1 to n. For some vertex v in T, the degree of v, denoted by $deg(v)$, is the number of edges incident to v. If $deg(v) = 1$, then v is called a *leaf*. Let P be a number sequence $[p_1, p_2, \ldots, p_i]$ and let x be a number. We denote the *concatenation* of P and x by $P \circ x$; i.e., $P \circ x = [p_1, p_2, \ldots, p_i, x]$.

According to Prüfer's proof, any sequence of $n - 2$ numbers, each number in $\{1, 2, \ldots, n\}$, can determine a unique labelled tree of n vertices. Such a number sequence is the Prüfer code of a labelled tree. Algorithm A is the straightforward implementation of Prüfer's proof.

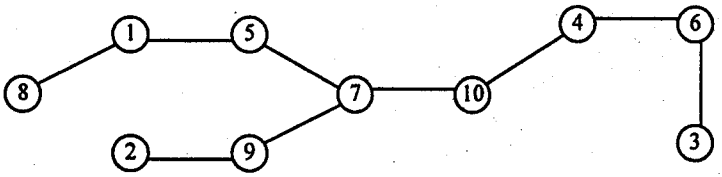

Fig. 2. A labelled tree.

Algorithm A

Input: A labelled tree T of n vertices.
Output: The Prüfer code of T.
Method:
Step 1. Let P be the null sequence.
Step 2. For $i = 1$ to $n - 2$ do
  Step 2.1. Let x be the leaf with the smallest number.
  Step 2.2. Remove x and its incident edge (x, y) from T.
  Step 2.3. $P := P \circ y$.
Step 3. P is the Prüfer code of T.

End of Algorithm

We give an example to illustrate Algorithm A. Let the input labelled tree T be the graph depicted in Figure 2. Initially, P is a null sequence. When $i = 1$, vertex 2 is the leaf with the smallest number. We remove vertex 2 and edge (2, 9) from T. Then $P \circ 9 = [9]$. When $i = 2$, vertex 3 is the leaf with the smallest number. Vertex 3 and edge (3, 6) are removed. Then $P \circ 6 = [9, 6]$. After the algorithm terminates, $P = [9, 6, 4, 10, 1, 5, 7, 7]$ which is the Prüfer code of T.

The most time-consuming step in Algorithm A is Step 2. Using a heap [10], the leaf with the smallest number can be found in $O(\log n)$ time. Hence, Step 2 takes $O(n \log n)$ time totally. The time-complexity of Algorithm A is also $O(n \log n)$.

The above method is used to understand Prüfer's mapping, but it is not optimal and is hard to parallelize. In the next section we propose an efficient algorithm to get the Prüfer code of a labelled tree, and our algorithm can be parallelized easily.

3. A Linear Time Algorithm for Generating the Prüfer Code of a Labelled Tree

Let T be a tree with n vertices numbered by $1, 2, \ldots, n$. First, rearrange T such that T is rooted at vertex n. Let T_v be a subtree of T which is induced by v and all of its descendants with root v. We then give each vertex a new label, called the *L-label*, as follows. For each vertex v of T, the L-label of v, denoted by $L(v)$, is the ordered pair $(M(v), S(v))$. Define $M(v)$ to be the maximum vertex in T_v and $S(v)$ to be the number of descendants w of v for which $M(w) = M(v)$. To compute M and S, we can use two postorder traversals [10]. Initially, let $S(v) = 0$ for $v = 1, 2, \ldots, n$. In the first postorder traversal, let $M(v)$ be the maximum of $M(v)$ and $M(w)$ for any child w of v.

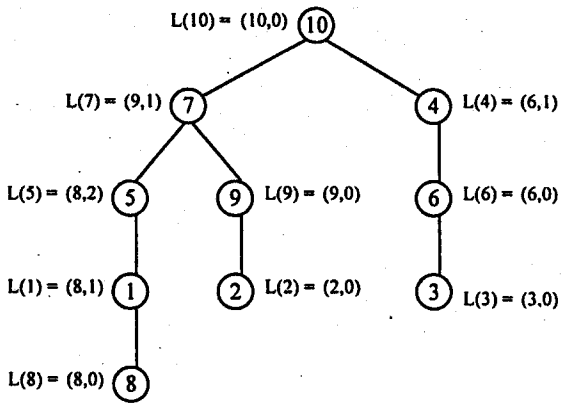

Fig. 3. A rooted labelled tree T with associated L-labels.

The second computes S using the postorder update: if $M(v) = M(w)$, then let $S(v)$ be $S(w) + 1$. The initial depth-first-search and the two postorder traversals each take time $O(n)$. We illustrate the L-labelling scheme by an instance. Graph T in Figure 3 is the tree rearranged from Figure 2 with root 10. To determine $L(7)$, we consider T_7. Since 9 is the maximum vertex in T_7, $M(7) = 9$. Besides, since vertex 7 has only one descendant whose first part of the L-label is 9, $S(7) = 1$. All L-labels of T are shown in Figure 3.

We say that $L(u)$ is lexicographically less than $L(v)$ if $M(u) < M(v)$ or if $M(u) = M(v)$ and $S(u) < S(v)$. In the above example, $L(8) < L(1)$ and $L(1) < L(10)$.

The following is our algorithm for generating the Prüfer code of a labelled tree, in which *parent*(v) stands for the parent of vertex v in T.

Algorithm B

Input: A labelled tree T of n vertices.
Output: The Prüfer code of T.
Method:
Step 1. Let P be the null sequence.
Step 2. Rearrange T such that T is rooted at vertex n. Note that the parent of each vertex is therefore determined. Let *parent*(n) be null.
Step 3. Compute $M(v)$ and $S(v)$ for all vertices v of T.
Step 4. Sort the vertices of T, using $M(v)$ as the primary key and $S(v)$ as the secondary key. Let the resulting sequence be $[L(v_1), L(v_2), \ldots, L(v_n)]$.
Step 5. For $i = 1$ to $n - 2$ do $P := P \circ parent(v_i)$.
Step 6. P is the Prüfer code of T.

End of Algorithm

We use graph T of Figure 3 again to illustrate Algorithm B. After doing Steps 1–3 as shown in Figure 3, sorting all L-labels lexicographically results in a sorted sequence [(2, 0), (3, 0), (6, 0), (6, 1), (8, 0), (8, 1), (8, 2), (9, 0), (9, 1), (10, 0)]. In Step 5, when

$i = 1, v_1$ is vertex 2 since (2, 0) is the L-label of vertex 2. Thus, $P \circ parent(2) = [9]$. When $i = 2$, v_2 is vertex 3 since (3, 0) is the L-label of vertex 3. Then $P \circ parent(3) = [9, 6]$. After Step 5 is terminated, $P = [9, 6, 4, 10, 1, 5, 7, 7]$ which is the Prüfer code of T.

It is obvious that Steps 1 and 6 of Algorithm B can be done in $O(1)$ time. In Step 2 the Breadth-First Search Algorithm [14] is applied to construct T_n in $O(n)$ time. Step 3 takes $O(n)$ time as described before. To sort all L-labels, we can apply the Distribution Counting Algorithm [18] which needs $O(n)$ time. Step 5 can be done trivially in $O(n)$ time. Thus, the complexity of Algorithm B is $O(n)$.

The correctness of Algorithm B is shown in the following lemma.

Lemma 3.1. *The resulting sequence P of Algorithm B is the Prüfer code of the input labelled tree.*

Proof. For each vertex x, $M(x) \geq x$. For any pair of nonroot vertex x and its parent y, $M(x) \leq M(y)$ and $L(x) < L(y)$. Thus, at the first iteration of Step 5, Algorithm B chooses the leaf with the smallest number. Then we delete the chosen vertex from the tree and update the current tree. By induction on the number of iterations, we can show that Algorithm B always chooses the smallest leaf of the current tree at each iteration. Assume on the contrary that at some iteration i, Algorithm B chooses a leaf x which is different from leaf y of the Prüfer code; i.e., at iteration i, y is the chosen vertex in Algorithm A, however, Algorithm B chose x. Since Algorithm A always chooses the leaf with the smallest number and Algorithm B chooses the leaf with the smallest L-label, $x > y$ and $L(x) < L(y)$. Moreover, since both x and y are leaves of the current tree, $M(x) < M(y)$. Denote the vertex $M(y)$ by z. Then it is clear that $M(z) = z$. (If $M(z) > z$, then $M(y) = z < M(z) \leq M(y)$. Contradiction.) It implies that $M(z) = z = M(y) > M(x)$ and so $L(z)$ is lexicographically greater than $L(x)$. This contradicts the fact that vertex z is deleted prior to x in Algorithm B. □

Therefore, we have the following theorem.

Theorem 3.2. *Algorithm B generates the Prüfer code of a labelled tree in $O(n)$ time.*

4. The Parallelization of Algorithm B

In this section we address the parallelization of Algorithm B, in which each step or substep can be implemented on the EREW PRAM model.

It is obvious that Steps 1, 5, and 6 of Algorithm B can be done in $O(1)$ time by using $O(n)$ processors. In Step 2 the parallel algorithm for rooting a tree is applied, which takes $O(\log n)$ time and $O(n/\log n)$ processors [11]. By parallel sorting algorithms [3], Step 4 takes $O(\log n)$ time with $O(n)$ processors. Step 3 is more complex, and we explain this step in detail in the following paragraphs with Figure 3 as an example for illustration.

Let $T = (V, E)$ be a labelled tree rooted at vertex n. First, we construct the Euler tour [6], 17] of T from the root and store the tour in an array $A = (a_1, a_2, \ldots, a_{2n-1})$. In our example, $A = (10, 7, 5, 1, 8, 1, 5, 7, 9, 2, 9, 7, 10, 4, 6, 3, 6, 4, 10)$. Constructing a Euler tour of a tree can be done in $O(1)$ time using $O(n)$ processors [17].

Table 1. Finding the leftmost and the rightmost appearances of all vertices.

	i																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a_i	10	7	5	1	8	1	5	7	9	2	9	7	10	4	6	3	6	4	10
b_i	0	1	2	3	4	3	2	1	2	3	2	1	0	1	2	3	2	1	0
f_i	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1

Let $l(v)$ and $r(v)$ be, respectively, the indices of the leftmost and rightmost appearances of v in A for any vertex v of T. The maximum number in T_v is equal to the maximum of $\{a_{l(v)}, a_{l(v)+1}, \ldots, a_{r(v)}\}$. To determine $l(v)$ and $r(v)$, we need another array $B = (b_1, b_2, \ldots, b_{2n-1})$ to keep the level of each element of A. The level of vertex v, denoted by $level(v)$, is the distance between v and the root in T. In our example, $B = (0, 1, 2, 3, 4, 3, 2, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, 1, 0)$. Array B can be determined in $O(1)$ time using $O(n)$ processors [17]. Element $a_i = v$ is the leftmost appearance of v in A if and only if $level(a_{i-1}) = level(a_i) - 1$, while $a_i = v$ is the rightmost appearance of v if and only if $level(a_{i+1}) = level(a_i) - 1$. For $i = 2, 3, \ldots, 2n-2$, let f_i be 1 if $level(a_{i-1}) = level(a_i) - 1$ or $level(a_{i+1}) = level(a_i) - 1$, and let f_i be 0, otherwise. Moreover, let f_1 and f_{2n-1} be 1. We can get $l(v)$ and $r(v)$ of all vertices by lexicographically sorting all pairs (a_i, i) of $f_i = 1$. This sorting takes $O(\log n)$ time and $O(n)$ processors. Tables 1 and 2 illustrate how to get $l(v)$ and $r(v)$.

After $l(v)$ and $r(v)$ of each vertex v are determined, we need to find the maximum of $\{a_{l(v)}, a_{l(v)+1}, \ldots, a_{r(v)}\}$. This can be done by applying the Basic Range Minima Algorithm [1], [6], [11] as follows. Define the *prefix maxima* of array $Z = (z_1, z_2, \ldots, zn)$ to be the array $(p_1, p_2, \ldots, p_n)$ such that $p_i = \max\{z_1, z_2, \ldots, z_i\}$, for $i = 1, 2, \ldots, n$. Similarly, define the *suffix maxima* of Z to be the array $(s_1, s_2, \ldots, s_n)$ such that $s_i = \max\{z_n, z_{n-1}, \ldots, z_i\}$, for $i = 1, 2, \ldots, n$. Then we build a complete binary tree T^* on the elements of array A (the Euler tour of T) such that each internal node u of T^* holds two arrays P_u and S_u, where P_u and S_u are the prefix maxima and the suffix maxima, respectively, of the leaves of T^*_u. Figure 4 shows the complete binary tree on the elements of array A of our example. Note that A is extended with $2^l - |A|$ zeros when $2^{l-1} < |A| < 2^l$ for some integer l. Moreover, an internal node is denoted by (x, y) where x is the layer number and y is the position at layer x on T^*. Constructing such a complete binary tree needs $O(\log n)$ time and $O(n)$ processors [11].

Since T^* is a complete binary tree, the lowest common ancestor of any two leaves can be found in constant time using a single processor [9]. For some vertex v of T,

Table 2. After sorting all pairs (a_i, i) of $f_i = 1$.

	v									
	1	2	3	4	5	6	7	8	9	10
$l(v)$	4	10	16	14	3	15	2	5	9	1
$r(v)$	6	10	16	18	7	17	12	5	11	19

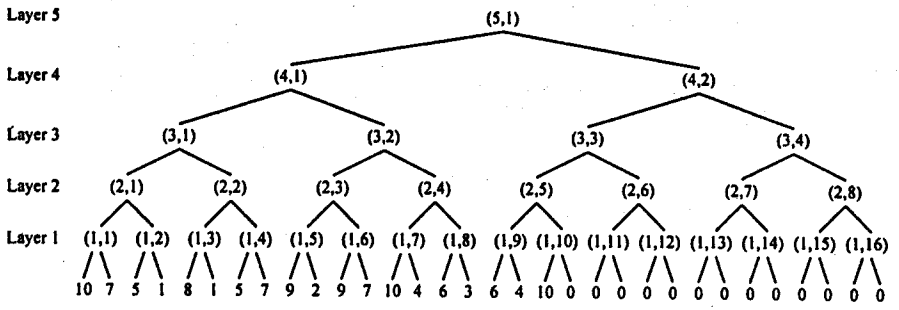

Fig. 4. The complete binary tree on the elements of array A.

if $l(v) = r(v)$, the maximum of $\{a_{l(v)}, a_{l(v)+1}, \ldots, a_{r(v)}\}$ is v. Suppose internal node (x, y) is the lowest common ancestor of $l(v)$ and $r(v)$ where $l(v) \neq r(v)$. Let s and t be the left and right children of (x, y), respectively. Then the leaves of $T_{(x,y)}$ are $a_{2^x\cdot(y-1)+1}, a_{2^x\cdot(y-1)+2}, \ldots, a_{2^x\cdot y}$ of A, and S_s is the suffix maxima of subarray $(a_{2^x\cdot(y-1)+1}, a_{2^x\cdot(y-1)+2}, \ldots, a_{2^x\cdot(y-1)+2^{x-1}})$ of A, and P_t is the prefix maxima of subarray $(a_{2^x\cdot(y-1)+2^{x-1}+1}, a_{2^x\cdot(y-1)+2^{x-1}+2}, \ldots, a_{2^x\cdot y})$ of A. The maximum of $\{a_{l(v)}, a_{l(v)+1}, \ldots, a_{r(v)}\}$ is the maximum of two elements: $s_{l(v)-2^x\cdot(y-1)}$ of S_s and $p_{r(v)-(2^x\cdot(y-1)+2^{x-1})}$ of P_t. Hence, determining $M(v)$ for a vertex v can be done in $O(1)$ time on T^* by a single processor. For example, we want to find the maximum number in T_7. Given $l(7) = 2$ and $r(7) = 12$, the lowest common ancestor of $l(7)$ and $r(7)$ is (4, 1). (See Figure 4.) The left and right children of (4, 1) are (3, 1) and (3, 2), respectively, in which $S_{(3,1)} = (10, 8, 8, 8, 8, 7, 7, 7)$ and $P_{(3,2)} = (9, 9, 9, 9, 10, 10, 10, 10)$. The maximum of $\{a_{l(7)}, a_{l(7)+1}, \ldots, a_{r(7)}\}$ is 9 since the second element of $S_{(3,1)}$ is 8 and the fourth element of $P_{(3,2)}$ is 9.

We claim that there is no read conflict on T^* when all vertices query their maximum numbers simultaneously. Let u and v be any two vertices of T. If the lowest common ancestor of $l(u)$ and $r(u)$ is different from the lowest common ancestor of $l(v)$ and $r(v)$, there is certainly no read conflict. Suppose they have the same lowest common ancestor, say node (x, y), in which s and t are the left and right children of node (x, y), respectively. Since $l(u)$ is different from $l(v)$, $l(u) - 2^x \cdot (y-1) \neq l(v) - 2^x \cdot (y-1)$. This implies that $l(u)$ and $l(v)$ correspond to different elements of S_s. With the same argument, $r(u)$ and $r(v)$ correspond to different elements of P_t. Therefore, all vertices can query their maximum numbers on T^* simultaneously without read conflict, and the first parts of all L-labels can be determined in $O(1)$ time using $O(n)$ processors.

After getting the first parts of all L-labels, we group vertices of T such that the vertices in each group have the same first part of the L-label. It is obvious that each group induces a path subgraph of T. Each group then uses the parallel prefix technique [11], [12] to count the number of vertices to get the second numbers. It can be done in $O(\log n)$ time using $O(n/\log n)$ processors. Figure 5 shows how to get the second part of each L-label.

The above procedure completes Step 3 of Algorithm B, and the complexity of Step 3 is $O(\log n)$ time with $O(n)$ processors on the EREW PRAM model.

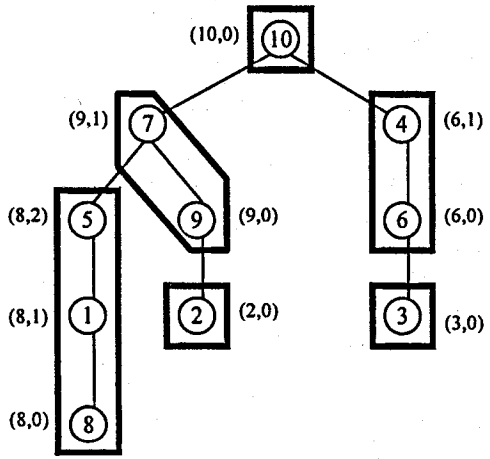

Fig. 5. Labelling the graph of Figure 3 in parallel.

5. Conclusion

The mapping between a labelled tree and its Prüfer code is interesting. In this paper we present an $O(n)$ time algorithm for generating a Prüfer code from a labelled tree. Our algorithm can be easily parallelized to solve this problem in $O(\log n)$ time using $O(n)$ processors on the EREW PRAM model.

Acknowledgment

The authors would like to thank the referees for their helpful comments which improved the quality of the paper.

References

[1] O. Berkman and U. Vishkin, Recursive Star-Tree Parallel Data Structure, Technical Report UMIACS-TR-90-40, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1990.
[2] A. Cayley, A Theorem on Trees, *Quartly Journal of Mathematics*, Vol. 23, 1889, pp. 376–378.
[3] R. Cole, Parallel Merge Sort, *SIAM Journal on Computing*, Vol. 17, No. 4, 1988, pp. 770–785.
[4] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986, Exercise 2, p. 666.
[5] Ö. Egecioglu and J. B. Remmel, Bijections for Cayley Trees, Spanning Trees, and Their q-Analogues, *Journal of Combinatorial Theory, Series A*, Vol. 42, 1986, pp. 15–30.
[6] H. N. Gabow, J. Bentley, and R. E. Tarjan, Scaling and Related Techniques for Geometry Problems, *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, Washington, DC, ACM Press, New York, 1984, pp. 135–145.
[7] R. Gould, *Graph Theory*, Benjamin/Cummings, Redwood City, CA, 1988.
[8] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
[9] D. Harel and R. E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, *SIAM Journal on Computing*, Vol. 13, No. 2, 1984, pp. 338–355.

[10] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, 3rd edn., Computer Science Press, Rockville, MD, 1990.
[11] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.
[12] C. P. Kruskal, L. Rudolph, and M. Snir, The Power of Parallel Prefix, *IEEE Transactions on Computers*, Vol. 34, No. 10, 1985, pp. 965–968.
[13] J. W. Moon, Various Proofs of Cayley's Formula for Counting Trees, in *A Seminar on Graph Theory* (F. Harary, Ed.), Holt, Rinehart, & Winston, New York, 1967, pp. 70–78.
[14] E. F. Moore, The Shortest Path Through a Maze. *Proceedings of the International Symposium on Switching Theory*, 1957, Part II, Harvard University Press, Cambridge, MA, 1959, pp. 285–292.
[15] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms for Computers and Calculators*, 2nd edn., Academic Press, New York, 1978, Exercise 46, p. 293.
[16] H. Prüfer, Neuer Beweis eines satzes über Permutationen, *Archiv der Mathematik und Physik*, Vol. 27, 1918, pp. 742–744.
[17] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplication and Parallelization, *SIAM Journal on Computing*, Vol. 17, No. 6, 1988, pp. 1253–1262.
[18] R. Sedgewick, *Algorithms*, 2nd edn., Addison-Wesley, Reading, MA, 1988.
[19] P. W. Shor, A New Proof of Cayley's Formula for Counting Labeled Trees, *Journal of Combinatorial Theory, Series A*, Vol. 71, No. 1, 1995, pp. 154–158.
[20] Y. L. Wang, H. C. Chen, and W. K. Liu, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 12, 1997, pp. 1236–1240.

Received October 2, 1998, *and in final form March* 1, 1999.