MOD-CHAR: An Implementation of Char's Spanning Tree Enumeration Algorithm and Its Complexity Analysis

R. JAYAKUMAR, K. THULASIRAMAN, SENIOR MEMBER, IEEE, AND M. N. S. SWAMY, FELLOW, IEEE

Abstract—**An implementation, called MOD-CHAR, of Char's spanning tree enumeration algorithm [3] is discussed. Two complexity analyses of MOD-CHAR are presented. It is shown that MOD-CHAR leads to better complexity results for Char's algorithm than what could be obtained using the straightforward implementation implied in Char's original presentation [3]. The class of graphs for which MOD-CHAR and, hence, Char's algorithm has linear time complexity per spanning tree generated is identified. This class is more general than the corresponding one identified in [7]. Using a result due to Matula [10], [12] on random graphs, it is proved that for almost all graphs MOD-CHAR has linear worst-case time complexity per spanning tree generated. It is also shown that for any complete graph MOD-CHAR requires, on the average, at most seven computational steps to generate a spanning tree. This result and computational experiences provide evidence to believe that for dense graphs of any order the time complexity of MOD-CHAR is $O(t)$, where t is the number of spanning trees generated. On the other hand, there is enough evidence to conclude that for sparse graphs, Char's original implementation is superior to MOD-CHAR.**

I. INTRODUCTION

ENUMERATING, that is, explicitly listing all the spanning trees of a graph is one of the extensively studied problems in graph theory. A classical result related to this problem is Kirchhoff's Matrix Theorem [9] which expresses the number of spanning trees of an n-vertex graph as a determinant of size $n-1$. A combinatorial proof of this theorem may be found in [13]. More recently, Chaiken and Kleitman [2] generalized Renyi's proof to include Tutte's generalization to digraphs [18]. The proofs of Renyi and Chaiken–Kleitman include variables in the matrix to yield explicit listing of the spanning trees. Although this approach yields an algorithm beating $O(n^{2.5})$ time for the number of spanning trees, the actual listing may take longer because the determinant calculation involves expressions rather than variables alone. Thus there has been a considerable interest in the design of efficient algorithms for spanning tree enumeration and a number of algorithms with varying efficiencies have been reported in the literature. The algorithm due to Gabow and Myers [6] is known to have the best possible asymptotic complexity of $O(nt)$ where t is the number of spanning trees of the n-vertex graph under consideration. This algorithm is essentially an efficient implementation of Minty's approach [11] for the enumeration problem. In 1968, Char [3] presented a conceptually simple algorithm for the spanning tree enumeration problem. Our recent studies [7], [8], show that Char's algorithm is computationally superior to Gabow and Myers' algorithm. However, further investigation of Char's algorithm resulting in more refined complexity results is called for. More recently, in [19] Winter presented yet another $O(nt)$ algorithm and has confirmed our earlier claim that Char's algorithm is superior to Gabow and Myers'.

An early motivation for studying the spanning tree enumeration problem came from electrical network theory where evaluating network functions using topological formulas was considered to be an attractive approach. In this and other applications which require explicit listing, any efficient spanning tree enumeration algorithm such as Gabow and Myers' will be an appropriate choice. There are applications where the spanning trees should be enumerated in a certain order. This is the case with the source-to-multiterminal network reliability analysis problem [14], [15] for which Gabow and Myers' algorithm is considered to be the best choice. There are also applications where only spanning trees with certain property need to be enumerated. For example, one solution to the transshipment problem in operations research [4] involves searching for a feasible spanning tree such that each circuit produced by adding a single edge has positive weight. For this problem the design of algorithms based on spanning tree enumeration is considered in [5]; and Gabow and Myers' algorithm may not be the most efficient choice in this case. The structure of Char's spanning tree enumeration algorithm [3] and our analysis of it in [7] suggest that it may yield an efficient algorithm for the transshipment problem. To elaborate on this, we first note that Char's algorithm essentially involves a lexicographic examination of tree and non-tree subgraphs. Making use of an initial spanning tree and a vertex numbering based on this tree, the algorithm reacts differently after trees and non-trees are generated so that two objectives are achieved. These are: (1) each subgraph is generated and tested in linear

Manuscript received April 8, 1985; revised April 8, 1986, May 17, 1987, and August 26, 1987. This paper was recommended by Associate Editor I. N. Hajj.

The authors are with the Faculty of Engineering and Computer Science, Concordia University, Montreal, P.Q., Canada H3G 1M8.

IEEE Log Number 8825002.

0098-4094/89/0200-0219$01.00 ©1989 IEEE

time and (2) not all of the $\Pi \deg(i)$ subgraphs are examined. The way the second objective is achieved is of interest to us in the context of designing an efficient algorithm for the transshipment problem. The structure of Char's algorithm is such that a number of non-tree subgraphs can be skipped without generation whenever a non-tree subgraph is encountered. In view of this, we expect to be able to incorporate in Char's algorithm a mechanism which would enable us to skip a number of infeasible trees whenever an infeasible tree is identified. If such a mechanism could be implemented efficiently, then a fast solution to the transshipment problem based on tree enumeration would become possible.

With such applications in mind, we continue in this paper our study of the spanning tree enumeration problem. Our objective here is to explore further the structure of Char's algorithm and to exploit this structure to design an implementation of the algorithm which leads to more refined complexity results. To make our presentation self-contained, we first review Char's algorithm in Section II and then present in Section III an implementation of Char's algorithm called MOD-CHAR. This implementation reduces the computation required per spanning tree by grouping together certain similar subgraphs for the purpose of testing. Next, in Section IV, we discuss the computational complexity of MOD-CHAR. Finally, in Section V we provide computational results comparing MOD-CHAR with Char's original implementation in [3]. For graph theory definitions and notations we follow [16].

II. Review of Char's Algorithm

Consider a connected undirected graph $G=(V, E)$ with $n=|V|$ vertices and $m=|E|$ edges. Let the vertices of G be denoted as $1,2,\cdots,n$. Let $\lambda=(\text{DIGIT}(1), \text{DIGIT}(2),\cdots,\text{DIGIT}(n-1))$ denote a $(n-1)$-digit sequence of vertices such that $\text{DIGIT}(i)$ is a vertex adjacent to vertex i in G. With each such sequence λ we can associate a subgraph $G_\lambda=(V_\lambda, E_\lambda)$ of G such that

$$V_\lambda=\{1,2,\cdots,n\},$$

and

$$E_\lambda=\{(1,\text{DIGIT}(1)),(2,\text{DIGIT}(2)),\cdots,(n-1,\text{DIGIT}(n-1))\}.$$

Noting that a spanning subgraph of G is a spanning tree of G if and only if it is connected and has $n-1$ edges, Char [3] characterizes sequences representing spanning trees as follows.

Tree Compatibility Property

The sequence $(\text{DIGIT}(1), \text{DIGIT}(2),\cdots,\text{DIGIT}(n-1))$ represents a spanning tree of an n-vertex graph G rooted at vertex n if and only if for each $j \leqslant n-1$ there exists a sequence of edges from $\{(i,\text{DIGIT}(i))\}$ forming a path from j to a vertex k with $k>j$. //

A sequence satisfying the tree compatibility property is called a *tree sequence*; otherwise it is a *non-tree sequence*. It can be shown that if $(\text{DIGIT}(1), \text{DIGIT}(2),\cdots, \text{DIGIT}(n-1))$ is a tree sequence representing a spanning tree T, then $\text{DIGIT}(i)$ is the neighbor of i in the path in T from i to n.

Char's algorithm first performs a search on G and finds a spanning tree called the *initial spanning tree*. This search could be either breadth-first search (BFS) or depth-first search (DFS) [1], [16]. During this search, the vertices of G are renumbered as $n, n-1,\cdots,1$ in the order in which they are visited. Let $\lambda_0=(\text{REF}(1), \text{REF}(2),\cdots, \text{REF}(n-1))$ be the sequence corresponding to the initial spanning tree. Clearly, $\text{REF}(i)>i$, $1 \leqslant i \leqslant n-1$. Starting with λ_0, the *initial tree sequence*, Char's algorithm enumerates all the other spanning trees of G by generating the sequences corresponding to the spanning trees of G. During this enumeration, the algorithm also generates certain non-tree sequences. We shall let t and t_0 denote, respectively, the number of tree sequences and the number of non-tree sequences generated.

Char's algorithm can be presented in ALGOL-like notation as follows.

Char's Algorithm to Enumerate all the Spanning Trees

```
procedure CHAR;
comment The graph G is represented by the adjacency
        lists of its vertices. REF(i) is the first entry
        in the adjacency list of vertex i, ADJ(i),
        and SUCC(DIGIT(i)) is the entry next to
        DIGIT(i) in ADJ(i).
begin
    find the initial spanning tree and obtain the
    initial tree sequence λ0 = (REF(1), REF(2),···,
    REF(n-1));
    renumber the vertices of the graph using the initial
    spanning tree;
    initialize DIGIT(i) := REF(i), 1 ⩽ i ⩽ n-1;
    output the initial spanning tree;
    k := n-1;
    while k ≠ 0 do begin
      if SUCC(DIGIT(k)) ≠ nil
        then begin
             DIGIT(k) := SUCC(DIGIT(k));
             if DIGIT(i), 1 ⩽ i ⩽ n-1, is a tree se-
             quence
               then begin
                    output the tree sequence;
                    k := n-1
               end
        end
        else begin
           DIGIT(k) := REF(k);
           k := k-1
        end
end
end CHAR;
```

Some important observations about the algorithm are now in order. For convenience, let $\text{LAST}(i)$ denote the last entry in $\text{ADJ}(i)$. To avoid complicated formalism in the following presentation, we assume that the graph G

under consideration has no vertex of degree one. This guarantees that $\mathrm{REF}(i) \neq \mathrm{LAST}(i)$, $1 \leqslant i \leqslant n-1$. Now let λ be the current sequence, with k as defined in the algorithm. Note that $\mathrm{DIGIT}(i) = \mathrm{REF}(i)$ for $i > k$, and that if a tree sequence (after λ_0) has been output then $\mathrm{DIGIT}(k) \neq \mathrm{REF}(k)$. Define $r \leqslant k+1$ by $r = k+1$ if $\mathrm{DIGIT}(k) \neq \mathrm{LAST}(k)$; otherwise r is the smallest integer such that $\mathrm{DIGIT}(i) = \mathrm{LAST}(i)$ for $r \leqslant i \leqslant k$. If $k = n-1$ and $\mathrm{DIGIT}(n-1) = \mathrm{LAST}(n-1)$, then let $s < n-1$ be the smallest integer such that $\mathrm{DIGIT}(i) = \mathrm{LAST}(i)$ for $s < i \leqslant n-1$; otherwise let $s = n-1$. Then

(i) If λ is a tree sequence, then the next sequence generated is $\mathrm{DIGIT}(1), \mathrm{DIGIT}(2), \cdots, \mathrm{DIGIT}(k), \mathrm{DIGIT}(k+1), \cdots, \mathrm{DIGIT}(s-1), \mathrm{SUCC}(\mathrm{DIGIT}(s)), \mathrm{REF}(s+1), \cdots, \mathrm{REF}(n-1)$.

(ii) If λ is not a tree sequence, then the next sequence generated is $\mathrm{DIGIT}(1), \mathrm{DIGIT}(2), \cdots, \mathrm{DIGIT}(r-2), \mathrm{SUCC}(\mathrm{DIGIT}(r-1)), \mathrm{REF}(r), \cdots, \mathrm{REF}(n-1)$.

Note that if λ is a tree sequence, then the next sequence generated differs from λ in every position $i \geqslant s$. But in this new sequence, $\mathrm{DIGIT}(i) = \mathrm{REF}(i) > i$ for every $i > s$. So this sequence is to be tested for the tree compatibility property only at position s. On the other hand, if λ is not a tree sequence, the next sequence differs from λ in every position i such that $r-1 \leqslant i \leqslant k$. But, again, in this new sequence $\mathrm{DIGIT}(i) = \mathrm{REF}(i) > i$, for $r \leqslant i \leqslant k$. So this sequence is to be tested for the tree compatibility property only at position $r-1$. Thus testing whether λ is a tree sequence or not involves testing the sequence for the tree compatibility property at exactly one position. The time for this test is bounded by n since it involves traversing at most n edges.

Let $T = \bigcup_{i=0}^{n-1} T_i$ where

(i) $T_0 = \{\lambda_0\}$, and

(ii) $T_i, 1 \leqslant i \leqslant n-1$, is the set of all the tree sequences such that i is the largest index for which $\mathrm{DIGIT}(i) \neq \mathrm{REF}(i)$.

Also let $T' = \bigcup_{i=1}^{n-1} T_i'$ where $T_i', 1 \leqslant i \leqslant n-1$, is the set of all the non-tree sequences such that i is the largest index for which $\mathrm{DIGIT}(i) \neq \mathrm{REF}(i)$. Note that $|T| = t$ and $|T'| = t_0$.

Let $G_k^{(s)}$, $1 \leqslant k \leqslant n-1$, be the graph obtained from G by coalescing the vertices $k, k+1, \cdots, n$ and let $T_k^{(s)}$ be the set of spanning trees of $G_k^{(s)}$. If $t(k) = |T_k^{(s)}|$, then it was shown in [7, theorem 3] that

$$t + t_0 = 1 + \sum_{k=1}^{n-1} (\deg(k) - 1) t(k) \tag{1}$$

where $\deg(k)$, $1 \leqslant k \leqslant n$, is the degree of vertex k in G.

An elegant formula for $t(k)$ is possible. Let G_i denote the weighted undirected graph obtained from G by assigning unit weight to each edge of G. Let the weighted graph $G_i, 2 \leqslant i \leqslant n-1$ be obtained from G_{i-1} by performing pivotal condensation, or equivalently star-delta transformation, at node $i-1$. The pivotal condensation operation at vertex $(i-1)$ in G_{i-1} is defined as follows. Let $w(i, j)$ denote the weight in G_{i-1} of the edge (i, j). If this edge is not in G_{i-1}, let $w(i, j) = 0$. Also let d_{i-1} denote the sum of the weights of all the edges in G_{i-1} incident on vertex $(i-1)$. Then, for each pair of vertices j_1 and j_2 adjacent to $i-1$, increase $w(j_1, j_2)$ by $w(i-1, j_1) * w(i-1, j_2)/d_{i-1}$. After all possible pairs of neighbors of vertex $(i-1)$ have been considered, delete from G_{i-1} the vertex $(i-1)$ and all the edges incident on it. The weighted graph that results is G_i. It can be shown [7] that

$$t(k) = d_1 d_2 \cdots d_{k-1}$$

and

$$t = d_1 d_2 \cdots d_{n-1}. \tag{2}$$

Using (2) we can rewrite (1) as

$$t + t_0 = 1 + t \sum_{k=1}^{n-1} \frac{\deg(k) - 1}{d_{n-1} d_{n-2} \cdots d_k}. \tag{3}$$

As regards the total number of computational steps required in the execution of Char's algorithm, consider a sequence $\lambda = (\mathrm{DIGIT}(1), \mathrm{DIGIT}(2), \cdots, \mathrm{DIGIT}(k-1), x, \mathrm{REF}(k+1), \cdots, \mathrm{REF}(n-1))$, with $x \neq \mathrm{REF}(k)$, generated by the algorithm. This sequence belongs to $T_k \cup T_k'$. To generate this sequence the algorithm explicitly requires setting $\mathrm{DIGIT}(i) = \mathrm{REF}(i)$ for each i, $k+1 \leqslant i \leqslant n-1$, in addition to setting $\mathrm{DIGIT}(k) = x$. Then λ is tested for the tree compatibility property at position k. Thus generating and testing λ involves the following two types of computational steps.

Type 1: $(n-k-1)$ steps to set $\mathrm{DIGIT}(i) = \mathrm{REF}(i)$, $k+1 \leqslant i \leqslant n-1$.

Type 2: C_k steps to set $\mathrm{DIGIT}(k) = x$ and to test λ for the tree compatibility property. (Note that $C_k \leqslant n$.)

Let COST1 and COST2 denote, respectively, the total numbers of Type 1 and Type 2 computations performed by Char's algorithm. It was shown in [7] that

$$\begin{aligned} \mathrm{COST1} &= \sum_{k=1}^{n-1} [t(k+1) - t(k)](n-k-1) \\ &= t \sum_{k=2}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k} - (n-2) \end{aligned} \tag{4}$$

and

$$\begin{aligned} \mathrm{COST2} &\leqslant n \sum_{k=1}^{n-1} |T_k U T_k'| \\ &= nt \sum_{k=1}^{n-1} \frac{\deg(k) - 1}{d_{n-1} d_{n-2} \cdots d_k}. \end{aligned} \tag{5}$$

If we let

$$H_n = \sum_{k=1}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k} \tag{6}$$

then it can be seen that

$$\mathrm{COST1} \leqslant H_n t \tag{7}$$

and

$$\text{COST2} \leqslant n^2 H_n t. \tag{8}$$

Thus we have the following theorem.

Theorem 1: Char's algorithm is of complexity $O(n^2 H_n t)$.
//

Note that H_n is dependent on the graph G as well as the initial spanning tree. However, as we will show in Section IV, in the case of certain classes of graphs, an appropriate initial spanning tree can be selected so that H_n becomes bounded above by a constant. More interestingly, we will show that $nH_n < 4$ for a complete graph on $n > 4$ vertices.

III. MOD-CHAR: An Implementation of Char's Algorithm

As we have seen in the previous section, Char's algorithm involves two types of computations, namely, Type 1 computations and Type 2 computations. Whereas the cost of Type 1 computations is $O(H_n t)$, Type 2 computations cost $O(n^2 H_n t)$ for an n-vertex graph. In this section we develop an implementation of Char's algorithm, which requires $O(nH_n t)$ Type 2 computations only. We call this implementation MOD-CHAR.

Consider an n-vertex undirected graph $G = (V, E)$. Let the vertices of G be numbered as in Char's algorithm. Consider a tree sequence $\lambda = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k), \text{REF}(k+1), \text{REF}(k+2), \cdots, \text{REF}(n-1))$ with $\text{DIGIT}(k) \neq \text{REF}(k)$, generated by Char's algorithm when applied on G. Note that λ is a tree sequence in T_k. It should be clear from the observations on Algorithm 1 made in the previous section that after generating λ, Char's algorithm proceeds to generate the tree sequences in $T_{k+1} \cup T_{k+2} \cup \cdots \cup T_{n-1}$ as well as the non-tree sequences in $T'_{k+1} \cup T'_{k+2} \cup \cdots \cup T'_{n-1}$ which have the same $\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k)$ as λ, by changing $\text{DIGIT}(n-1), \text{DIGIT}(n-2), \cdots, \text{DIGIT}(k+1)$ in an appropriate manner. Then another sequence in $T_k \cup T'_k$ is generated by setting $\text{DIGIT}(i) = \text{REF}(i)$ for $k+1 \leqslant i \leqslant n-1$, and changing $\text{DIGIT}(k)$ in λ. Thus the sequences in $T_k \cup T'_k$ having the same $\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1)$ as λ are not generated immediately after λ, and generating each one of these sequences requires at most n Type 2 computations. We now show how these computations can be reduced by an appropriate implementation of Char's algorithm.

Consider a tree sequence $\lambda_k = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), \text{REF}(k), \text{REF}(k+1), \cdots, \text{REF}(n-1))$. First we examine how Char's algorithm generates a sequence in $T_k \cup T'_k$ from λ_k. Let G'_k denote the forest obtained by removing the edge $(k, \text{REF}(k))$ from the spanning tree G_k corresponding to λ_k. In G'_k the vertices $k+1, k+2, \cdots, n$ are in one component and the vertex k is in the other component. For each vertex $x \neq \text{REF}(k)$ adjacent to vertex k, the sequence $\lambda''_k = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), x, \text{REF}(k+1), \cdots, \text{REF}(n-1))$ will be in $T_k \cup T'_k$. This sequence can be classified as follows. If vertices k and x are in the same component of G'_k, then there is a circuit passing through vertex k in G''_k (the subgraph of G corresponding to the sequence λ''_k), and so λ''_k is a non-tree sequence in T'_k. On the other hand, if vertices k and x are in different components of G'_k, then λ''_k is a tree sequence in T_k. Thus if, for each λ_k defined as above, we have the information whether each neighbor x of k in G is in the same component of G'_k as vertex k or not, then only one comparison is required to test each sequence of the form $\lambda''_k \in T_k \cup T'_k$. Clearly we need to obtain this information for only those neighbors $x < k$.

In order to collect the above information, we associate a label $\text{LABEL}(i)$ with each vertex $i < k$ of G'_k. For each neighbor $x < k$ of k, $\text{LABEL}(x)$ is computed such that $\text{LABEL}(x) = k$ if the vertices k and x are in the same component of G'_k; and $\text{LABEL}(x) = n$ otherwise. In order to obtain these label values, we traverse the path in G'_k from vertex x to either vertex k or to some vertex greater than k. If this path leads to vertex k, then we set $\text{LABEL}(x) = k$; otherwise $\text{LABEL}(x) = n$. These computations are performed efficiently as follows.

First we initialize $\text{LABEL}(i) = 0$, $1 \leqslant i \leqslant k-1$, and $\text{LABEL}(k) = k$. For each neighbor x of k such that $x < k$, we traverse the path in G'_k from x to some vertex y for which either $\text{LABEL}(y) \neq 0$ or $y > k$. If $y > k$, then for every vertex $v \neq y$ on this path we set $\text{LABEL}(v) = n$; otherwise we set $\text{LABEL}(v) = \text{LABEL}(y)$. It is easy to see that this procedure determines the label values correctly. Moreover, each one of the edges $(1, \text{DIGIT}(1)), (2, \text{DIGIT}(2)), \cdots, (k-1, \text{DIGIT}(k-1))$ is traversed at most twice and hence this traversal requires at most $2(k-1)$ computational steps. Initialization requires k computational steps. Thus the label values corresponding to λ_k can be computed in $O(n)$ time. Once the label values are computed, testing each sequence in $T_k \cup T'_k$ corresponding to λ_k would require one computational step. Thus Char's algorithm when implemented as above will require considerably less number of Type 2 computations. We now present a recursive version of this implementation.

MOD-CHAR to Enumerate all the Spanning Trees

```
procedure MOD-CHAR;
comment procedure MOD-CHAR enumerates all the
        spanning trees of a connected n-vertex graph
        using algorithm MOD-CHAR. The graph G is
        represented by the adjacency lists ADJ(i), 1 ≤ i
        ≤ n − 1, of its vertices.
procedure GENERATE(k);
comment procedure GENERATE, when called with the
        argument k, sets DIGIT(k) to generate a tree
        sequence. This procedure uses a local array
        LABEL.
begin
    if k = n
        then output the tree sequence
      else begin
          {Set DIGIT(i) to REF(i), k ≤ i ≤ n − 1}
          DIGIT(k) := REF(k);
          GENERATE(k + 1);
```

```
        {Generate all the sequences in T_k ∪ T_k' having
        the same
        DIGIT(1), DIGIT(2), ···, DIGIT(k−1)}
        compute LABEL(x) for each neighbor x < k
        of k;
        for x ∈ ADJ(k)-REF(k) do
        if LABEL(x) = n or x > k
        then begin
          DIGIT(k) := x;
          GENERATE(k+1)
        end
    end
end GENERATE;
begin
    find the initial tree sequence (REF(1), REF(2), ···,
    REF(n−1));
    renumber the vertices of G;
    GENERATE(1)
end MOD-CHAR;
```

IV. Computational Complexity of MOD-CHAR

In this section we study the computational complexity of MOD-CHAR. Since MOD-CHAR is essentially a new implementation of Char's algorithm, all the complexity results we develop here for MOD-CHAR hold true for Char's algorithm too. Also, in the rest of the paper, by Char's algorithm we refer to the implementation implied in Char's original presentation [3]. In our analysis of MOD-CHAR, all the computations included in the analysis of Char's algorithm are also included. To output t spanning trees, at least $(n-1)t$ computational steps are needed. Also we need $O(m+n)$ computations to find the initial spanning tree, where m is the number of edges in the graph. We do not include these computations in our complexity analyses. Furthermore, we assume, without loss of generality, that the graph G is biconnected with minimum degree at least three.

It is easy to see that MOD-CHAR requires the same number of Type 1 computations as Char's algorithm. So it follows from (7) that COST1 for MOD-CHAR is $O(nt)$, since $H_n \leqslant n$.

To evaluate the cost of Type 2 computations for MOD-CHAR, we first note that with each tree sequence of the form $\lambda_k = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), \text{REF}(k), \text{REF}(k+1), \cdots, \text{REF}(n-1))$ we can associate the set $S(\lambda_k)$ of sequences of the form $(\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), x, \text{REF}(k+1), \text{REF}(k+2), \cdots, \text{REF}(n-1))$, where $x \neq \text{REF}(k)$ is a vertex adjacent to k. Clearly $S(\lambda_k) \subseteq T_k \cup T_k'$. The Type 2 computations performed to generate and test $S(\lambda_k)$ consist of two components—the computations to determine the label values and the computations to generate and test each sequence in $S(\lambda_k)$. Determining label values requires $O(n)$ computations. Once label values are determined, generating and testing a sequence in $S(\lambda_k)$ requires constant time. So the cost $\text{COST2}(\lambda_k)$ of the Type 2 computations required to test all the sequences in $S(\lambda_k)$ is $O(n)$. Furthermore, COST2 for MOD-CHAR is given by

$$\text{COST2} = \sum_{k=1}^{n-1} \sum_{\lambda_k \in S_k} \text{COST2}(\lambda_k) \tag{9}$$

where S_k is the collection of all sequences of the form $(\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), \text{REF}(k), \text{REF}(k+1), \cdots, \text{REF}(n-1))$.

We now present two analyses to evaluate COST2. Our first analysis relates COST2 to a topological parameter associated with G.

We assume that the initial spanning tree used in MOD-CHAR has been generated by performing a DFS on G. We shall denote this tree by T_{DFS}. The choice of a DFS tree as the initial spanning tree will result in no loss of generality, since Char's algorithm and MOD-CHAR can be started with any spanning tree as the initial spanning tree. As in Char's algorithm, we renumber the vertices of G as $n, n-1, \cdots, 1$ in the order in which they are visited during the DFS. For each $i, i \leqslant n-1$, we define $\text{NEXT}(i)$, $\text{LEAF}(i)$, γ_i, and σ_i as follows.

$\text{NEXT}(i) = \max\{\, j \mid j \leqslant i$ and j is adjacent to at least two vertices greater than $j \,\}$;

$\text{LEAF}(i) = \max\{\, j \mid j$ is a leaf and $j \leqslant i \,\}$;

$\gamma_i = 1 +$ length of the path in T_{DFS} from i to $\text{NEXT}(i)$;

$\sigma_i = 1 +$ length of the path in T_{DFS} from i to $\text{LEAF}(i)$.

Let

$$\gamma = \max_{i \leqslant n-1} \{\gamma_i\},$$

$$\sigma = \max_{i \leqslant n-1} \{\sigma_i\}.$$

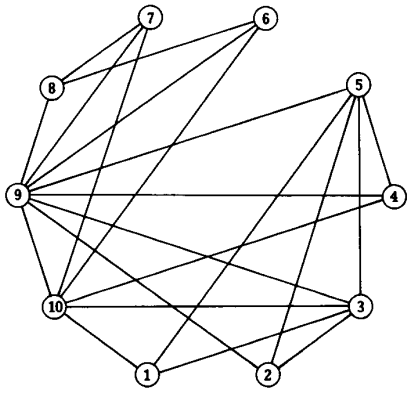
To illustrate the above definitions, consider the graph G in Fig. 1(a). The vertices of G are numbered using the DFS tree shown in Fig. 1(b). In Fig. 2 we show the $\text{NEXT}(i)$, $\text{LEAF}(i)$, γ_i, and σ_i values.

The following theorem establishes a bound for COST2 in terms of γ.

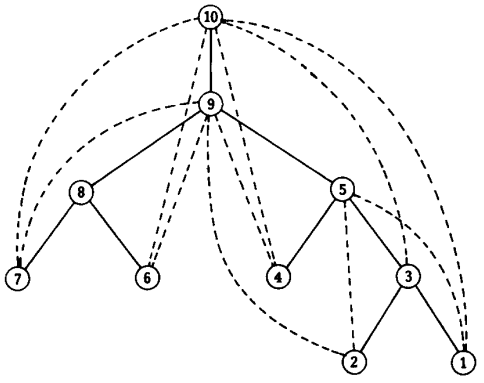
Theorem 2: For MOD-CHAR, COST2 is $O(\gamma nt)$.

Proof: Our proof involves associating each $\lambda_k = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), \text{REF}(k), \text{REF}(k+1), \cdots, \text{REF}(n-1))$ with an appropriate tree sequence λ so that we can charge $\text{COST2}(\lambda_k)$ to λ.

If $S(\lambda_k)$ consists of at least one tree sequence, then $\text{COST2}(\lambda_k)$ can be charged to that sequence. If not, let $r < k$ be the highest number such that for $r < i \leqslant k$ there are no tree sequences of the form $(\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(i-1), y, \text{REF}(i+1), \cdots, \text{REF}(n-1))$, $y \neq \text{REF}(i)$ and there is at least one tree sequence $\lambda' = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(r-1), z, \text{REF}(r+1), \text{REF}(r+2), \cdots, \text{REF}(n-1))$ with $z \neq \text{REF}(r)$. In this case, all the sequences in each one of the sets $S(\lambda_i)$, $r < i \leqslant k$, where λ_i is of the form $(\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(i-1), \text{REF}(i), \cdots, \text{REF}(n-1))$



(a)



(b)

Fig. 1. (a) Graph g. (b) Depth-first search spanning tree.

i	NEXT(i)	LEAF(i)	γ_i	σ_i
1	1	1	1	1
2	2	2	1	1
3	3	2	1	2
4	4	4	1	1
5	4	4	2	2
6	6	6	1	1
7	7	7	1	1
8	7	7	2	2
9	7	7	3	3

Fig. 2. NEXT (i), LEAF (i), γ_i, σ_i values for the graph of Fig. 1.

will be tested before the tree sequence $\lambda' \in S(\lambda_r)$ is identified. As we have already pointed out, the cost COST2(λ_i), $r \leqslant i \leqslant k$ of Type 2 computations to test all the sequences in $S(\lambda_i)$ is $O(n)$. So, if we charge to λ' all the costs COST2(λ_i), $r \leqslant i \leqslant k$, then the total cost charged to λ' will be

$$\sum_{i=r}^{k} \text{COST2}(\lambda_i) = O((k-r+1)n).$$

From the definition of NEXT(k) it can be seen that $r \geqslant \text{NEXT}(k)$ and so $k-r+1 \leqslant \gamma_k$. So it follows that the tree sequence λ' will be charged $O(\gamma_k n)$ Type 2 computations. Thus each tree sequence will be charged with at most $O(\gamma n)$ computations and the theorem follows. //

An important observation is now in order. According to the charging technique used in the proof of the above theorem, if $S(\lambda_k)$ has more than one tree sequence, then exactly one of these tree sequences will be charged with cost $O(n)$ and the cost charged to each one of the other tree sequences in $S(\lambda_k)$ is equal to zero.

Since, by assumption, each vertex is adjacent to at least three vertices, and leaves are adjacent to only higher numbered vertices, it follows that $\gamma_i \leqslant \sigma_i$ and so $\gamma \leqslant \sigma$. This leads to the following.

Corollary 2.1: For MOD-CHAR, COST2 is $O(\sigma nt)$. //

Since COST1 is $O(nt)$ and $\gamma \geqslant 1$, we get the following from Theorem 2.

Theorem 3: MOD-CHAR is of complexity $O(\gamma nt)$. //

Corollary 3.1: MOD-CHAR is of complexity $O(\sigma nt)$. //

Let $A_i = \{j | j > i$ and vertex j is adjacent to vertex $i\}$. If $|A_i| \geqslant 2$, then it can be seen that $\gamma_i = 1$. Let M denote the class of all graphs such that the vertices of each graph in M can be numbered as in Char's algorithm with the property that $|A_i| \geqslant 2, 1 \leqslant i \leqslant n-2$. A numbering with this property is called an *M-numbering*. Then for every graph in M, $\gamma = 2$, since $\gamma_i = 1$ of all $i, 1 \leqslant i \leqslant n-2$, and $\gamma_{n-1} = 2$. So, for all graphs in M, MOD-CHAR is of complexity $O(nt)$. Thus we get the following.

Theorem 4: MOD-CHAR is of complexity $O(nt)$ for every graph G which permits an M-numbering. //

Consider next the class M' of all n-vertex biconnected graphs which have maximum degree $n-1$. A vertex with degree $n-1$ will be called a star vertex. Let G be any graph in M' and S be a star tree of G. Assigning number n to the star vertex in S and the number $n-1$ to any other vertex of S, we can obtain an M-numbering of S. If this were not possible, then there would exist a subset $X = \{x_1, x_2, \cdots, x_k\}$ of vertices such that vertices n and $n-1$ are not in X and each x_i is adjacent to exactly one vertex (namely, the vertex n) not in X. But then, in such a case, the vertex n would be a cut vertex of G, contradicting that G is biconnected. Thus $M' \subseteq M$ and we get the following from Theorem 4.

Corollary 4.1: For an n-vertex biconnected graph with maximum degree $n-1$, MOD-CHAR is of complex $O(nt)$. //

It was proved in [7] that Char's algorithm is of complexity $O(nt)$ in the cases of the complete graph, the ladder, and the wheel. Since $M' \subseteq M$ contains these graphs, it follows that MOD-CHAR is of complexity $O(nt)$ for a larger class of graphs than those identified in [7]. In fact, later in this section, we prove a more interesting result for the complete graphs.

Next, we present our second complexity result, in which we shall relate COST2 to H_n.

Theorem 5: COST2 for MOD-CHAR is $O(nH_n t)$ and hence MOD-CHAR is of complexity $O(nH_n t)$.

Proof: As already noted, for each $\lambda_k = (\text{DIGIT}(1), \text{DIGIT}(2), \cdots, \text{DIGIT}(k-1), \text{REF}(k), \text{REF}(k+1), \cdots, \text{REF}(n-1))$, $\text{COST2}(\lambda_k)$ is $O(n)$. So, it follows from (9) that $\text{COST2} = O(n\Sigma t(k))$. By (2), $\Sigma t(k) = H_n t$. So, $\text{COST2} = O(nH_n t)$ and MOD-CHAR is of complexity $O(nH_n t)$. //

It can be seen that for any graph G which permits an M-numbering, $d_i \geq 2$ for all i, $1 \leq i \leq n-2$, and so in this case

$$H_n \leq 1 + \sum_{k=1}^{n-2} \frac{1}{2^k} < 2$$

thereby confirming Theorem 4. However, we can prove a stronger result.

Theorem 6: MOD-CHAR is of complexity $O(nt)$ in the case of an n-vertex biconnected graph G which has a $(1+\log_2 n)$-vertex connected subgraph permitting an M-numbering.

Proof: For the graph G defined in the theorem

$$\begin{aligned} nH_n &= n \sum_{k=1}^{n-1} \frac{1}{d_{n-1} d_{n-2} \cdots d_k} \\ &< n \sum_{k=0}^{\log_2 n} \frac{1}{2^k} + n \sum_{k=1}^{n-2-\log_2 n} \frac{1}{n2^k} \\ &< 2n+1 \\ &< 3n. \end{aligned}$$

Thus the proof follows from Theorem 5. //

According to a result due to Matula [10], [12] on random graphs, almost all graphs have a complete graph of order $\log_2 n$ for large n. This property along with Theorem 6 leads to the following interesting result.

Theorem 7: For almost all graphs MOD-CHAR is of complexity $O(nt)$. //

We have earlier proved that MOD-CHAR is of complexity $O(nt)$ for an n-vertex complete graph. Now we prove a stronger and more interesting result using Theorem 5. Since for an n-vertex complete graph $\deg(k) = n-1$ for all k, we get from (1) that

$$t + t_0 = 1 + (n-2)tH_n.$$

Since $t + t_0 < 2t$ for a complete graph ([7, corollary 6.1]) it follows that

$$H_n < \frac{2}{n-2}. \tag{10}$$

From (10) and Theorem 5, we can see that MOD-CHAR has $O(t)$ time complexity for a complete graph.

Now we determine an upper bound for the number of computational steps required by MOD-CHAR to generate a spanning tree of a complete graph. Note that each Type 1 computation involves setting $\text{DIGIT}(k) = \text{REF}(k)$ for some k and hence one assignment operation. Thus the total number of assignments for all the Type 1 computations is $H_n t - (n-1)$. For a given k, to find the label values we require $2(k-1) \leq 2n-4$ computational steps. Thus the label computations require at most $(2n-4)H_n t$ computational steps. Moreover, $t + t_0 < 2t$ computational steps are required to generate all the sequences for a complete graph. Thus at most $2t + (2n-3)H_n t$ computational steps are required by MOD-CHAR for an n-vertex complete graph. From this observation and (10) we can show that MOD-CHAR requires, on the average, less than

$$6 + \frac{2}{n-2}$$

computational steps to generate a spanning tree of an n-vertex complete graph. Thus we get the following.

Theorem 8: MOD-CHAR requires, on the average, at most 7 computational steps to generate a spanning tree of a complete graph of any order. //

We now make a few comments on the complexity results given in Theorems 3 and 5. The complexity $O(nH_n t)$ is less pessimistic than the complexity $O(\gamma nt)$ in the sense that the total number of computational steps (both Types 1 and 2) required by MOD-CHAR will be no more than $4nH_n t$. However, the constant γ for a given T_{DFS} is much easier to compute than H_n. So Theorem 3 will be useful to obtain a quick estimate of the complexity of MOD-CHAR. We wish to point out that in all the examples considered H_n was found to be less than γ. For example, for the graph of Fig. 1, $H_n \approx 0.91$ whereas $\gamma = 3$. These results once again highlight the influence of the initial spanning tree on the complexity of MOD-CHAR. Even though selecting an initial spanning tree leading to the minimum number of computational steps is difficult, heuristics to generate an initial spanning tree aimed at reducing the number of computational steps can be developed. These heuristics should attempt to minimize γ and/or H_n. In fact, the heuristics presented in [7] attempted to achieve a value of 2 for γ and H_n.

V. Computational Results

In this section we present a computational evaluation of MOD-CHAR in comparison to the implementation envisaged by Char in his paper [3]. We do not provide a comparison with Gabow and Myers' algorithm since our study in [8] and the recent study by Winter [19] show that Char's algorithm is superior to Gabow and Myers'.

Char's implementation and MOD-CHAR both require two types of operations: Type 1 and Type 2. We may recall that Type 1 computations are those required to generate the sequences and Type 2 computations are those required to test the sequences for the tree compatibility property. Generating a sequence requires accessing entries (or equivalently edges) from the adjacency lists, whereas the test for the tree compatibility property requires comparisons and traversals of edges in a subgraph. Since edge traversal also essentially involves accessing edges from the adjacency lists, we consider *edge access* (which includes edge traversals and comparisons) as the basic operation

TABLE I
COMPUTATIONAL RESULTS

Number of Vertices	Density	Average Number Computational Steps per Tree	
		CHAR	MOD-CHAR
8	0.50	4.616	6.165
8	0.61	4.500	6.128
8	0.72	5.226	6.220
8	0.82	4.831	6.169
8	1.00	5.734	5.682
9	0.70	4.785	5.587
9	0.75	4.881	5.656
9	0.80	4.982	5.703
9	0.90	5.408	5.666
9	1.00	6.128	5.742
10	0.70	5.308	5.732
10	0.75	5.463	5.712
10	0.80	5.570	5.731
10	0.85	5.700	5.730
10	0.90	6.066	5.755
10	1.00	6.520	5.785
11	0.82	6.712	5.787
11	1.00	6.884	5.786

performed by both Char's implementation and MOD-CHAR. In order to obtain an evaluation which is independent of such factors as the programming language, data structures used, etc., our comparison of these two implementations will be in terms of the average number of edge accesses performed by them. First, we shall highlight the salient features of these two implementations.

Both MOD-CHAR and Char's implementation require the same number of Type 1 computations. They differ only in the way the test for the tree compatibility property, namely the Type 2 computations, is performed.

Consider a graph G. For a given value of $k \leqslant n-1$, let, as defined in Section III, G_k be the subgraph corresponding to a sequence λ_k. For each neighbor $x < k$ of vertex k, the tree compatibility test is to be performed. To perform this test, Char's implementation requires traversing the path in G_k starting at vertex x. There is no provision in this implementation to avoid traversal of a path or a segment of a path more than once. This means that if, for example, the paths starting at, say, three neighbors x_1, x_2, x_3 of k overlap, then those edges in the common segment of these paths will be traversed *three times*. On the other hand, MOD-CHAR performs the tree compatibility test through label values. In this implementation it is ensured that no edge in G_k is traversed *more than twice*, even when overlapping paths are present.

For a sparse graph, the probability of having more than two overlapping paths which start at lower numbered neighbors of k is very small. So, for such graphs, the number of edge accesses required by Char's implementation will not likely be more than those required by MOD-CHAR. Furthermore, the number of non-tree subgraphs which contain trivial circuits of the type (k, x), (x, k) will be quite large. Char's implementation would require exactly two edge accesses to detect the presence of each of these trivial circuits. On the other hand, there is no provision in MOD-CHAR to detect the presence of these circuits, except through label computations. For example, we can show that in the case of a 10-vertex circuit, Char's implementation would require, on the average, 22 edge accesses per spanning tree generated, whereas the corresponding number for MOD-CHAR is about 44. These features of Char's implementation lead us to conclude that for sparse graphs it is computationally superior to MOD-CHAR.

In the case of dense graphs, the probability of having more than two overlapping paths which start at lower numbered neighbors of k is very high. So, for such graphs, it is more likely that an edge in G_k will be traversed *more than twice* in Char's implementation, thereby resulting in more number of edge accesses than those required by MOD-CHAR. In Table I we present computational results for several test graphs with varying densities, where the density of an n-vertex graph with m edges is defined [19] to be equal to $2m/(n(n-1))$. These test graphs were generated randomly as in [17]. The following points emerge from these results.

1) Performance of MOD-CHAR gets better as graphs become denser and denser. In the case of a 9-vertex graph, as the density varies from 0.7 to 1, the average number of edge accesses performed by Char's implementation increases from 4.785 to 6.128—an increase of about 28 percent. The corresponding increase for 10-vertex graphs is about 22 percent. On the other hand, these increases for MOD-CHAR are merely about 2.7 and 1 percent, respectively.

2) The average number of edge accesses required by Char's implementation varies from 4.785 for a 9-vertex graph with density 0.7 to 6.884 for an 11-vertex complete graph—an increase of about 40 percent. The corresponding increase for MOD-CHAR is only about 3.5 percent.

3) For complete graphs on 8 or more vertices, MOD-CHAR outperforms Char's implementation, since by Theorem 8 for these graphs, MOD-CHAR requires no more than 7 edge accesses per tree generated.

4) For a 10-vertex graph with density = 0.90, MOD-CHAR requires 5.755 edge accesses per tree generated, whereas Char's implementation requires 6.066. For denser and larger graphs, there is virtually no increase for MOD-CHAR, whereas the number continually increases for Char's implementation. This result demonstrates that for large dense graphs, the number of edge accesses required by MOD-CHAR becomes proportional to the number of trees generated and is independent of the number of vertices in the graph.

From the above we can see that MOD-CHAR will be computationally superior to Char's implementation for large and dense graphs. This conclusion is also easily justifiable using Theorem 8 and the fact that for large

dense graphs, the dominating term in the expression for H_n is $1/d_{n-1}$. As the graph becomes denser d_{n-1} tends to increase (for a complete graph $d_{n-1} = n/2$) and the term nH_n in the complexity expression becomes a constant for large n.

VI. Summary and Conclusion

In this paper we have presented and studied the computational complexity of Char's spanning tree enumeration algorithm. Two complexity analyses of this implementation, called MOD-CHAR, have been discussed. These analyses show that MOD-CHAR achieves better worst-case complexity results for Char's algorithm than what could possibly be obtained using the implementation implied in Char's original paper [3]. The class of graphs for which the complexity of MOD-CHAR is $O(nt)$ has been identified. This class is more general than the corresponding one identified in [7]. Using a result due to Matula on random graphs [10], [12], it has been shown that for almost all graphs, the complexity of MOD-CHAR or equivalently Char's algorithm is $O(nt)$. In fact, we have shown that for complete graphs, MOD-CHAR and hence Char's algorithm requires, on the average, at most seven computational steps per spanning tree generated. This property has not been observed for any of the other spanning tree enumeration algorithms which explicitly list the trees one at a time.

Whereas MOD-CHAR achieves better complexity results for Char's algorithm, it is pointed out that for sparse graphs Char's implementation as implied in [3] will be superior to MOD-CHAR. From computational results on randomly generated dense graphs, we conclude that for large dense graphs MOD-CHAR will be superior to Char's implementation. These results along with the result in Theorem 8 show that for large dense graphs the complexity of MOD-CHAR and hence that of Char's algorithm is $O(t)$.

Our results clearly demonstrate the influence the initial spanning tree has on the complexity of MOD-CHAR. Efficient heuristics for selecting an initial spanning tree (as developed in [7] for Char's implementation) which aim at minimizing the value of γ and/or H_n can be developed, leading to a further speedup of MOD-CHAR.

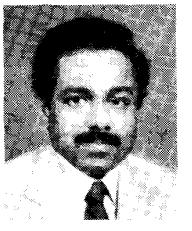
A few graph-theoretic problems remain to be solved. One of these is that of selecting a spanning tree with the smallest value of γ. We conjecture that for all biconnected graphs with minimum degree at least 3, a spanning tree with $\gamma = 2$ is possible. In other words, we conjecture that all such graphs permit an M-numbering, so that for these graphs H_n is bounded above by a constant.

Acknowledgment

The authors thank the reviewers for their constructive comments which have resulted in considerable improvement in the quality of the paper.

References

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[2] S. Chaiken and D. J. Kleitman, "Matrix tree theorems," *Journal of Comb. Theory*, Ser. A, vol. 24, no. 3, pp. 377–381, May 1978.
[3] J. P. Char, "Generation of trees, two-trees and storage of master forests," *IEEE Trans. Circuit Theory*, vol. CT-15, pp. 228–238, Sept. 1968.
[4] V. Chvátal, *Linear Programming*. New York: Freeman, 1983.
[5] G. Finke and J. H. Ahrens, "Enumeration of basic and basic feasible solutions of the transportation problem," Dep. Appl. Math., Tech. Univ. of Nova Scotia, Halifax, Canada, unpublished.
[6] H. N. Gabow and E. W. Myers, "Finding all spanning trees of directed and undirected graphs," *SIAM J. Comput.*, vol. 7, no. 3, pp. 280–287, Aug. 1978.
[7] R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy, "Complexity of computation of a spanning tree enumeration algorithm," *IEEE Trans. Circuits Syst.*, vol. CAS-31, pp. 853–860, Oct. 1984.
[8] R. Jayakumar and K. Thulasiraman, "Analysis and study of a spanning tree enumeration algorithm" in *Combinatorics and Graph Theory*, Springer-Verlag Lecture Notes in Math., no. 833, pp. 284–289, 1980.
[9] G. Kirchhoff, "Uber die Auflosung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Strome gefuhrt wird," *Ann. Phys. Chem.*, vol. 72, pp. 497–508, 1847. (English translation in *IRE Trans. Circuit Theory*, vol. CT-5, pp. 4–7, 1958.)
[10] D. W. Matula, "The largest clique size in a random graph," Tech. Rep. CS-76-08, Dep. of Computer Science, Southern Methodist Univ., Dallas, TX, 1976.
[11] G. J. Minty, "A simple algorithm for listing all the trees of a graph," *IEEE Trans. Circuit Theory*, vol. CT-12, p. 120, Mar. 1965.
[12] E. M. Palmer, *Graphical Evolution: An Introduction to the Theory of Random Graphs*. New York: Wiley-Interscience, 1985, pp. 75–78.
[13] A. Renyi, "On the Enumeration of Trees," in *Combinatorial Structures and their Applications*, R. Guy, H. Hanani, N. Sauer, and J. Schonheim, Eds., New York: Gordon Breach, 1970, pp. 355–360.
[14] A. Satyanarayana and A. Prabhakar, "New topological formula and rapid algorithm for reliability analysis of complex networks," *IEEE Trans. Reliability*, vol. R-27, pp. 82–100, June 1978.
[15] A. Satyanarayana, "A unified formula for the analysis of some network reliability problems," *IEEE Trans. Reliability*, vol. R-31, pp. 23–32, Apr. 1982.
[16] M. N. S. Swamy and K. Thulasiraman, *Graphs, Networks, and Algorithms*. New York: Wiley-Interscience, 1981.
[17] G. Tinhofer, "On the generation of random graphs with given properties and known distribution," in "*Graphs, Data Structures, Algorithms*," in *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 265–297, 1979.
[18] W. T. Tutte, "The dissection of equilateral triangles into equilateral triangles," in *Proc. Cambridge Phil. Soc.*, vol. 44, pp. 203–217, (1948).
[19] P. Winter, "An algorithm for the enumeration of spanning trees," *BIT*, vol. 26, pp. 44–62, 1986.



R. Jayakumar received the B.E. (Hons.) degree in electronics and communication engineering from the University of Madras, Madras, India, in 1977, the M.S. degree in computer science from the Indian Institute of Technology, Madras, India in 1980, and the Ph.D. degree in engineering and computer science from Concordia University, Montreal, Canada, in 1984.

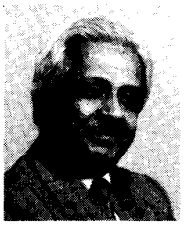
Since 1984 he has been an Assistant Professor of Computer Science at Concordia University, Montreal, Canada. His research interests are in VLSI algorithms and architectures, fault-tolerant VLSI systems, VLSI design automation, and graph theory and graph algorithms.

Dr. Jayakumar is a member of the Association for Computing Machinery.

✠

K. Thulasiraman (M'72–SM'84) received the B.E. and M.Sc. (Engg.) degrees from the University of Madras in 1963 and 1965 and the Ph.D. degree from the Indian Institute of Technology, Madras, in 1968.

During 1965–1981 he was with the Departments of Electrical Engineering and Computer Science at the Indian Institute of Technology, Madras. Since 1982 he has been with Concordia University where he is a Professor of Electrical and Computer Engineering. He has co-authored the book *Graphs, Networks and Algorithms* (Wiley-Interscience, 1981) (Russian translation by Mir Publishers, Moscow, in 1984). Recently, he visited the Tokyo Institute of Technology supported by a senior fellowship from the Japan Society for Promotion of Science. His current research interests are in graph theory, parallel and distributed computations and applications.



M. N. S. Swamy (S'59–M'62–SM'74–F'80) received the B.Sc. (Hon.) degree in mathematics from Mysore University, India, in 1954, the Diploma in electrical communication engineering from the Indian Institute of Science, Bangalore, India, in 1957, and the M.Sc. and Ph.D. degrees in electrical engineering from the University of Saskatchewan, Canada, in 1960 and 1963, respectively.

He was Chairman of the Electrical Engineering Department, Concordia University, Montreal, from 1970 until August 1977 when he became Dean of the Faculty of Engineering and Computer Science at the same university, a position he still holds. His research interests include number theory, semiconductor circuits, and network theory. He has published a number of papers on these subjects and also co-authored the book *Graphs, Networks and Algorithms* (Wiley-Interscience, 1981). A Russian translation of this book was published by Mir Publishers, Moscow, in 1984.

Dr. Swamy is a member and fellow of several professional societies and has served actively in a number of capacities in the IEEE. He is a co-recipient of the IEEE-CAS 1986 Guillemin–Cauer best paper award with Dr. Roytman and Dr. Plotkin.