A TREE CONVOLUTION ALGORITHM

FOR THE SOLUTION OF QUEUEING NETWORKS*

Simon S. Lam and Y. Luke Lien**

TR-165

January 1981
December 1981 (1st revision)
May 1982 (2nd revision)

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

* To appear in Communications of the ACM. This work was supported by National Science Foundation Grant No. ECS78-01803.

**Y. Luke Lien's current address: IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

Abstract

A new algorithm called the tree convolution algorithm for the computation of normalization constants and performance measures of product-form queueing networks is presented. The algorithm is very efficient, compared to existing algorithms, in the solution of networks with many service centers and many sparse routing chains. (A network is said to have sparse routing chains if the chains visit, on the average, only a small fraction of all centers in the network.) In such a network, substantial time and space savings can be achieved by exploiting the network's routing information. The time and space reductions are made possible by two features of the algorithm: (1) the sequence of array convolutions to compute a normalization constant is determined according to the traversal of a tree; and (2) the convolutions are performed between arrays that are smaller than arrays used by existing algorithms. The routing information of a given network is used to configure the tree to reduce the algorithm's time and space requirements; some effective heuristics for optimization are described herein. An exact solution of a communication network model with 64 queues and 32 routing chains is illustrated.

Key words and phrases: queueing networks, queueing theory, product-form solution, computational algorithms, tree convolution algorithm, sparse routing chains, performance evaluation.

1. INTRODUCTION

Queueing networks have been widely and successfully used in the modeling of computer systems and communication networks.* Presently, most known networks that are analytically tractable belong to the class of BCMP networks which have a product-form solution [2]. The product-form solution gives the improper equilibrium probabilities of network states. These improper probabilities need to be divided by a normalization constant to form a proper probability distribution. The normalization constant is given by the sum of the improper probabilities over all feasible network states. For a network consisting of only open routing chains with constant arrival rates, the summation yields a simple closed-form expression for the normalization constant. For other networks (such as those with closed chains and those with chain population size constraints [10]) the time and space computational requirements of the normalization constant may be very large due to the large number of feasible network states present in any non-trivial model.

The convolution algorithm for product-form queueing networks was first discovered by Buzen [4] for single-chain networks and extended by Chandy, Herzog and Woo [5] and by Reiser and Kobayashi [22] to multi-chain networks. Consider a network of M service centers with K closed routing chains. Let N_k be the population size of chain k. The convolution algorithm encounters difficulties when the chain population sizes in $\underline{N} = (N_1, N_2, \ldots, N_k)$ become large or when K becomes large. First, when chain population sizes become large, the normalization constant $G(\underline{N})$ may become too large (causing a floating point overflow) or too small (causing a floating point underflow) [7, 19]. A dynamic scaling technique to

---

*See the September 1978 special issue of ACM Computing Surveys on queueing network models of computer system performance. A survey of queueing network models of computer communication networks is available in [15, 26].

solve this problem was recently proposed [11]. Second, the algorithm's time and space requirements increase exponentially with K; more specifically, they are proportional to $\prod_{k=1}^{K} (N_k+1)$. Hence, the algorithm is not applicable to networks with more than a few chains.

The mean value analysis (MVA) algorithm of Reiser and Lavenberg [23] bypasses the evaluation of $G(\underline{N})$ and computes the performance measures of mean queue lengths and chain throughputs directly. It avoids the problem of floating point overflows. (Floating point underflows may still occur [20].) However, its time and space requirements also grow exponentially with K.

Other computational algorithms available (such as LBANC and CCNC in [7,24] and NCA in [21]) are variants of the basic convolution and MVA algorithms and thus also suffer from the exponential growth in space and time requirements as K increases.

The modeling of distributed systems and communication networks often require the use of a large number of routing chains in the model. The time and space requirements are so large that none of the previously mentioned algorithms is applicable. Various approximate solution techniques based upon the convolution algorithm [27] or based upon a mean value analysis [1, 6, 16, 20, 25] have been proposed for such models as well as models involving large chain population sizes.

We present in this paper a new computational algorithm based upon convolutions, to be called the tree convolution algorithm or the tree algorithm.

The algorithm exploits information on the sets of centers visited by chains (routing information) which has not been heretofore utilized by other algorithms. Such exploitation can give rise to very substantial savings in computational time

and space requirements for networks with many centers and many routing chains that visit, on the average, only a small fraction of all centers in the network (sparseness property). In a network with the sparseness property, if the chains are also clustered in certain parts of the network (locality property) then the computational time and space requirements can be further reduced.

Both the sparseness and locality properties are often present in models of large communication networks and distributed systems. For example, consider the modeling of a store-and-forward packet switching network. Such a network typically has tens of store-and-forward nodes. Each node has several queues, one for each communication channel connecting the node to a neighboring node. The network provides virtual channels from external packet sources to external packet sinks. The virtual channels are flow-controlled and are modeled by closed routing chains [15, 20]. Each such closed chain typically traverses just a few communication channels from its source to its destination. In a 1973 ARPANET measurement study, the average path length of packets was measured to be 3.24 communication channels [9]. Hence, the network has very sparse routing chains. The locality property is also evident from the observed phenomena of distance-dependence of traffic, incest, favorite sites and such, described in [9].

Several new ideas are present in the tree algorithm. First, the sequence of array convolutions to compute a normalization constant is determined by the traversal of a tree whose leaf nodes correspond to service centers in the network model. Second, the concept of partial covering of chains by a subset of centers is introduced. As a result, convolutions are performed between arrays that are smaller than the K-dimensional arrays used by existing algorithms. The routing information of a given network is utilized to construct the tree; tree construction heuristics are designed with the objective of minimizing the tree convolution

algorithm's space and time requirements. A tree data structure also facilitates different space-time tradeoffs for different networks and the incorporation of storage management techniques for the solution of very large networks.

Summary of this paper

In Section 2, some definitions and the notation for product-form queueing networks are reviewed. In Section 3, the basic ideas of tree traversal and array convolutions are discussed and illustrated with a small example. A preprocessor for the tree algorithm is then described. The preprocessor has two functions: (1) use the routing information of a given network to construct a tree, and (2) calculate the algorithm's time and space requirements for a given tree prior to tree traversals and array convolutions. In Section 4, the computation of network performance measures is discussed. Time-space tradeoffs of the algorithm as well as storage management considerations are addressed. In Section 5, a high-level description of the entire algorithm is presented. In Section 6, an exact solution of a communication network model with 64 queues and 32 routing chains is illustrated.

2. DEFINITIONS AND NOTATION

Consider a BCMP network with M service centers and K closed routing chains. Let N_k denote the population size of chain k. The network population vector is

$$\underline{N} = (N_1, N_2, \ldots, N_K).$$

The normalization constant for this network population vector is $G(\underline{N})$.

Let n_{mk} denote the number of chain k customers in center m. Define the network state

$$\underline{n} = (\underline{n}_1, \underline{n}_2, \ldots, \underline{n}_m)$$

where

$$\underline{n}_m = (n_{m1}, n_{m2}, \ldots, n_{mK}) \qquad m = 1,2,\ldots,M.$$

The product-form solution for the equilibrium probability of network state $\underline{n}$ is [2]

$$P(\underline{n}) = \frac{\prod_{m=1}^{M} p_m(\underline{n}_m)}{G(\underline{N})} \tag{1}$$

where

$$p_m(\underline{n}_m) = \left[\prod_{i=1}^{n_m} \frac{1}{\mu_m(i)} \right] n_m! \prod_{k=1}^{K} \frac{\rho_{mk}^{n_{mk}}}{n_{mk}!} \tag{2}$$

where

$$n_m = n_{m1} + n_{m2} + \ldots + n_{mK}$$

$$\rho_{mk} = \lambda_{mk} \tau_{mk}$$

where τ_{mk} is the mean service time of a chain k customer in center m (assuming that he is served at the rate of 1 second of work required per second) and λ_{mk} is the relative arrival rate of chain k customers to center m determined by the routing behavior of chain k. (See [2] for details.) Finally, $\mu_m(i)$ is the service rate of center m when it has a total of i customers. A center is said to be queue-dependent if $\mu_m(i)$ varies with i. A service center is said to be fixed-rate if $\mu_m(i) = 1$ for all $i \geq 0$. For simplicity and without loss of generality, we shall omit the possibility of service rate dependence on the numbers of customers in a center belonging to different chains, that is permitted in [2]. The reader is also referred to [2] for a description of the 4 types of service centers in BCMP networks.

The normalization constant $G(\underline{N})$ is by definition

$$G(\underline{N}) = \sum_{\substack{\underline{n} \text{ such that} \\ \sum_{m=1}^{M} \underline{n}_m = \underline{N}}} \prod_{m=1}^{M} p_m(\underline{n}_m) \tag{3}$$

The real-valued function p_m, for $m=1,2,\ldots,M$, has the domain $\{(j_1,j_2,\ldots,j_K) \mid 0 \leq j_k \leq N_k,\ k=1,2,\ldots,K\}$ and can be represented by a K-dimensional array indexed between $\underline{0}$ and $\underline{N}$, where $\underline{0}$ is a K-vector of all zeroes. The convolution of two such functions, say p_1 and p_2, defines a real-valued function, say g_2, over the same domain, as follows

$$g_2(\underline{i}) = \sum_{j_1=0}^{i_1} \cdots \sum_{j_K=0}^{i_K} p_1(\underline{j}) p_2(\underline{i}-\underline{j}) \qquad \text{for } \underline{0} \leq \underline{i} \leq \underline{N} \tag{4}$$

where the binary relation $\leq$ between two vectors is satisfied for each pair of corresponding components in the vectors.

In shorthand notation, (4) will be written as

$$g_2 = p_1 \circledast p_2 = p_2 \circledast p_1$$

Define

$$g_m = g_{m-1} \circledast p_m \qquad m = 2,3,\ldots,M \tag{5}$$

where g_1 is p_1 by definition. Note that the normalization constants for network population vectors between $\underline{0}$ and $\underline{N}$ are contained in the array g_M. Specifically $G(\underline{N})$ defined by (3) is given by $g_M(\underline{N})$.

Equations (4) and (5) define the convolution algorithm [5, 22] and have a space requirement of the order of $2 \prod_{k=1}^{K} (N_k+1)$ and a time requirement of the order of $(M-1) \prod_{k=1}^{K} [(N_k+1)(N_k+2)/2]$.

For a network of fixed-rate service centers, (4) and (5) reduce to

$$g_m(\underline{i}) = g_{m-1}(\underline{i}) + \sum_{k=1}^{K} \rho_{mk}\, g_m(\underline{i} - \underline{1}_k) \text{ for } \underline{0} \leq \underline{i} \leq \underline{N} \qquad (6)$$

where $\underline{1}_k$ is a K-vector with the k^{th} component equal to one and all others equal to zero, $g_m(\underline{0}) = 1$ by definition and $g_m(\underline{i} - \underline{1}_k)$ is zero if $i_k = 0$. The convolution operation described by (6) is sometimes referred to as feedback filtering [22]. Its space requirement to compute g_M is of the order of $\prod_{k=1}^{K} (N_k+1)$ and its time requirement is of the order of $MK \prod_{k=1}^{K} (N_k+1)$. Each unit in the above space requirements is an array location. Each unit in the above time requirements corresponds approximately to the execution time of one multiplication and one addition.

Note that given the functions p_m for m = 1,2,...,M, both (5) and (6) apply the convolution operation to the functions sequentially one after another. We shall therefore refer to such an algorithm as a sequential convolution algorithm.

3. KEY ELEMENTS OF THE ALGORITHM

The key ideas and observations that motivated the algorithm's development are first discussed in Sections 3.1 and 3.2. A small example is presented for illustration in Section 3.3. In Section 3.4, a preprocessor for the tree algorithm is described. Time and space requirements are discussed in Section 3.5.

3.1 Partially Covered Arrays

Consider routing chain k. Let CENTERS(k) be the set of service centers visited by chain k. Let SUBNET denote a subset of the M service centers. With respect to SUBNET, chain k is said to be fully covered if CENTERS(k) $\subseteq$ SUBNET; chain k is said to be noncovered if the intersection of CENTERS(k) and SUBNET is null; otherwise, chain k is said to be partially covered.

Let SUBNET = $\{m_1, m_2, \ldots, m_s\} \subseteq \{1,2,\ldots,M\}$.

Define

$$g_{SUBNET} = p_{m_1} \circledast p_{m_2} \circledast \ldots \circledast p_{m_s}.$$

Suppose that the array g_{SUBNET} has been computed as an intermediate step towards the computation of the network normalization constant $G(\underline{N})$ for population vector $\underline{N}$. The key observation here is that if some chains are noncovered or fully covered with respect to SUBNET, then only some of the elements in the array g_{SUBNET} are needed for the computation of $G(\underline{N})$; the amount of space required to store the necessary elements in g_{SUBNET} may be made substantially less than $\prod_{k=1}^{K} (N_k+1)$ locations.

Partition the set of K chains into the following 3 sets with respect to SUBNET:

σ_{pc} = {k|chain k is partially covered by SUBNET}

σ_{fc} = {k|chain k is fully covered by SUBNET}

σ_{nc} = {k|chain k is noncovered by SUBNET}

Now note that only those elements of g_{SUBNET} with index values in the following set are needed for further convolutions to arrive at $G(\underline{N})$.

$$\{\underline{i} = (i_1,\ldots,i_K) \quad | \quad i_k = 0,\ldots,N_k \quad \text{if } k \in \sigma_{pc},$$
$$i_k = N_k \quad \text{if } k \in \sigma_{fc},$$
$$i_k = 0 \quad \text{if } k \in \sigma_{nc} \}$$

Let $|\sigma|$ denote the cardinality of set σ. For the purpose of computing $G(\underline{N})$, it is sufficient to store g_{SUBNET} as an array with dimensionality $|\sigma_{pc}|$ indexed by $\underline{i}_{pc} = \{i_k, k \in \sigma_{pc}\}$. Such an array is termed a <u>partially covered array</u>. The amount of space needed for a partially covered array is $\prod_{k \in \sigma_{pc}} (N_k+1)$ locations. (Additionally, a small amount of space is also needed to store σ_{pc}.)

For queueing networks with properties of sparseness and locality, the space savings from the use of partially covered arrays instead of K-dimensional arrays

can be very substantial. A programming language that provides for dynamic allocation of storage for arrays (such as PL/I) will facilitate the implementation of partially covered arrays. However, it is often possible to realize much of the space savings of partially covered arrays even with static storage allocation (see the discussion on space requirements of the network example in Section 6).

Let SUBNET be partitioned into two subsets, SUBNET1 and SUBNET2. We then have

$$g_{SUBNET} = g_{SUBNET1} \circledast g_{SUBNET2} \tag{7}$$

Chain k is said to be overlapped if it is partially covered with respect to SUBNET1 and also SUBNET2. Partition the set of K chains into 4 sets, σ_{00}, σ_{01}, σ_{10} and σ_{11}. A chain belongs to one of the 4 sets depending upon its status with respect to SUBNET (partially covered or not) and its status with respect to SUBNET1 and SUBNET2 (overlapped or not), such as shown in Table 1.

If partially covered arrays are employed for the convolution operation in (7), then the time requirement of (7) is

$$\text{time(SUBNET1, SUBNET2)} = \prod_{k\varepsilon\sigma_{10}\cup\sigma_{01}} (N_k+1) \prod_{k\varepsilon\sigma_{11}} \frac{(N_k+2)(N_k+1)}{2} \tag{8}$$

Equation (8) gives the actual number of multiplications required for (7). Almost the same number of additions are also needed for (7); specifically, $\prod_{k\varepsilon\sigma_{01}\cup\sigma_{11}} (N_k+1)$ fewer additions are needed than multiplications. We shall use (8) as a measure of the time requirement of the convolution operation in (7). Each time unit in (8) is interpreted to be the time needed to execute 1 multiplication and 1 addition. (A derivation of (8) is given in Appendix I.)

We have shown that with the use of partially covered arrays, the convolution operation in (7) can be performed with very substantial time and space savings

when there are few partially covered chains in SUBNET1, SUBNET2 and SUBNET. Given a subset of centers in a network that has many centers and sparse routing chains, it is highly likely that only a few chains will be partially covered by the subset.

3.2 Ordering of Array Convolutions

Consider now the sequential convolution algorithm defined by (5). The algorithm begins with the subnet {1} consisting of center 1 and then sequentially "merge" the subnet with other service centers one after another. The algorithm ends when all centers have been merged. Partially covered arrays can be employed to implement the sequential convolution algorithm and realize some time and space savings. However, if we are free to merge service centers into small subnets, and small subnets into large subnets in any order, we can achieve substantially more time and space savings than a sequential algorithm. The objective is to find a sequence of mergers to minimize the number of partially covered chains in intermediate subnets by exploiting routing information.

An implementation of the above idea using a binary tree is next described. Place the service centers at the leaf nodes of the tree. (See Fig. 1.) Each node in the tree corresponds to a subset of service centers (a subnet) that are descendents of that node. Thus, the root node corresponds to the entire network. Visit all nodes in the tree according to some order of tree traversal. The root node is visited last. A branch node may be visited only after its two sons have both been visited. When a branch node is visited, its g array is computed from the g arrays of the node's sons using (7). (The g array of a leaf node is defined below.) Finally, when the root node is visited, the normalization constant $G(N)$ for the whole network is obtained.

Note that the sequential convolution algorithm is a special case of the tree convolution algorithm. It corresponds to the tree shown in Fig. 2. With both the

sequential algorithm and a general tree algorithm, the number of convolutions required to compute $G(\underline{N})$ is M-1. The tree algorithm, however, permits greater flexibility for reducing the size of partially covered arrays by exploiting routing information (see Section 3.2 below).

Unless otherwise stated, we shall refer to the sequential algorithm with the implicit assumption that K-dimensional arrays are implemented; and we shall refer to the general tree algorithm with the implicit assumption that partially covered arrays are implemented.

The g arrays for the leaf nodes (individual service centers) are evaluated using a modification of (2). Let {m} denote a subnet consisting of center m only, σ_{pc} denote its set of partially covered chains, and σ_{fc} denote its set of fully covered chains. The g array of {m} is given by

$$g_{\{m\}}(\underline{i}_{pc}) = \left[\prod_{j=1}^{n_m} \frac{1}{\mu_m(j)} \right] n_m! \prod_{k\varepsilon\sigma_{pc}} \frac{\rho_{mk}^{i_k}}{i_k!} \prod_{k\varepsilon\sigma_{fc}} \frac{\rho_{mk}^{N_k}}{N_k!} \qquad (9)$$

$$\text{for } i_k = 0,1,\ldots,N_k,\ k\varepsilon\sigma_{pc}$$

where the product over the set σ_{fc} is equal to 1 if σ_{fc} is void, and

$$n_m = \sum_{k\varepsilon\sigma_{pc}} i_k + \sum_{k\varepsilon\sigma_{fc}} N_k.$$

The computation of (9) requires $4 \prod_{k\varepsilon\sigma_{pc}\cup\sigma_{fc}} (N_k+1)$ multiplications.

The g array of a subnet consisting of two leaf nodes can be obtained using the recursion in (6) if one of the leaf nodes corresponds to a fixed-rate service center. The time requirement of (6) is less than (8) if the population sizes of overlapped chains are large. (See Appendix II.)

3.3 An Example

Consider a network of 4 centers, each consisting of a fixed-rate server. Suppose that there are 4 closed routing chains. The number of customers in each chain is 2. The (relative) traffic intensities ρ_{mk} for m = 1,2,3,4 and k = 1,2,3,4 are shown in Table 2.

Suppose that the service centers are placed at the leaf nodes of a binary tree as shown in Fig. 3 and postorder tree traversal is adopted. The set of partially covered chains at a node is shown next to it in Fig. 3. Note that chain 4 is fully covered by {3}. There are 3 mergers altogether. The set of overlapped chains for each merger and a list of fully covered chains after a merger are shown in Table 3.

The g arrays at leaf nodes are shown in Table 4. The convolutions and g arrays at the two branch nodes are shown in Table 5.

Finally, the normalization constant $G(\underline{N})$ where $\underline{N} = (2,2,2,2)$ is given by the convolution

$$G(\underline{N}) = g_{\{1,2\}}(0)\, g_{\{3,4\}}(2) + g_{\{1,2\}}(1)\, g_{\{3,4\}}(1) + g_{\{1,2\}}(2)\, g_{\{3,4\}}(0)$$

$$= 27.984375.$$

3.4 A Preprocessor for Constructing a Tree

A closed product-form queueing network is completely specified by its traffic intensities $\{\rho_{mk}\}$, service rate functions $\{\mu_m\}$, population vector $\underline{N}$ and routing information {CENTERS(k)}. Given such information, the time and space needed by the tree algorithm to compute a normalization constant depend upon the sequence of mergers of subnets (a merger corresponds to an array convolution). The merger sequence is determined by the tree configuration, the placement of centers at leaf nodes and the tree traversal order. It is easy

to see that any merger sequence can be specified by specifying just the tree (both its configuration and the placement of centers) with the tree traversal order fixed. We have adopted the use of postorder tree traversals. The construction of a tree with the objective of minimizing the time and space requirements of subsequent tree traversals and array convolutions will be referred to as tree planting. No efficient algorithm has been found to solve such an optimization problem. We have, however, found many efficient and effective heuristics [17, 18].

The tree algorithm employs two procedures. The first procedure, to be referred to as the preprocessor, is used for planting trees and evaluating the time and space requirements (needed to compute specified performance measures) for planted trees. The second procedure performs the main function of the tree algorithm, namely, tree traversals and array convolutions (for the computation of performance measures). The preprocessor has time and space requirements that are much smaller than the requirements of array convolutions (see below). In this paper, the time and space requirements of the tree algorithm refer only to the requirements of the second procedure.

Note that the tree algorithm's computational requirements are different for different networks. Given a network, the preprocessor provides us with accurate a priori estimates that can be compared with the requirements of other computational algorithms; more importantly, we can determine if the requirements are feasible for the computer being used.

We have investigated many heuristic procedures for tree planting. An experimental study of the tree algorithm's time and space requirements as well as a family of effective tree planting procedures are presented in [17, 18]. The basic algorithm that is common to all procedures in the family is given below.

```
Algorithm 1 {basic tree planting procedure}

begin
    initialization;
    while at least two subnets are present do
    begin
        perform superset merger;
        sort subnets according to a size criterion;
        select two subnets for merger according to a cost criterion;
        merge the selected subnets into one
        {comment:  a tree node is formed}
    end
end
```

Initially, there are M subnets with each center constituting a subnet (a leaf node). In general, the algorithm to determine the sequence of mergers is as follows. First, it checks for superset relationships between subnets. A superset relationship exists if the set of partially covered chains of a subnet contains the set of partially covered chains of another subnet. Subnets with superset relationships are merged. In the absence of superset relationships, two subnets are selected for the next merger on the basis of a cost criterion. The selection is facilitated by first sorting subnets according to a size criterion.

Many cost and size criteria have been proposed and studied experimentally [17, 18]. We next describe the criteria that were used by the tree algorithm to solve the numerical example in Section 6. The size criterion used is first described. Let SUBNET be a subset of centers and σ_{pc} be the set of chains partially covered by SUBNET. The weight of SUBNET is defined to be

$$\text{weight(SUBNET)} = \sum_{k \in \sigma_{pc}} |\text{CENTERS}(k) - \text{SUBNET}|$$

where the notation $|A-B|$ is the number of elements that are in set A and not in set B.

Given that the tree planting procedure selects the first candidate for the next merger by the weight criterion such that the heaviest subnet is selected,

the other candidate for the next merger is then selected to minimize a cost function to be defined. Suppose that subnet A has been selected and subnet B is a prospective partner. The cost of a merger of the 2 subnets is calculated as follows. For every partially covered chain in B, its status in A is checked and a cost is calculated. There are 3 possible cases.

Case 1. The chain is not covered by A. The cost of the chain is +1.

Case 2. The chain is partially covered by A but not fully covered by A $\cup$ B. The cost of the chain is -1.

Case 3. The chain is partially covered by A and fully covered by A $\cup$ B. The cost of the chain is -2.

Define the dimension of a subnet to be the number of partially covered chains in it. Note that the change in the dimension of A caused by a partially covered chain in B following a merger with B is equal to +1, 0 and -1 respectively for the above 3 cases. Instead of using the dimension changes, 0 and -1, as the costs for case 2 and case 3 respectively, we found that the use of smaller costs (-1 and -2) made the tree planting procedure much more effective [17, 18].

The specific tree planting procedure that was used for the numerical example in Section 6 is presented below. It plants a balanced binary tree and skips the step for superset mergers. The number M of centers must be a power of 2. Initially, each center constitutes a subnet (leaf node) at the lowest level of the tree. The tree is then constructed one level at a time.

Algorithm 2 {procedure to plant a balanced tree}

```
begin
    initialization;
    for each level of the tree from the leaves to the root do
    begin
        sort subnets by weight in decreasing order;
        mark all subnets;
        while some subnets are marked do
        begin
            choose the heaviest marked subnet as the first candidate for
              the next merger;
            choose from among the remaining marked subnets the other candidate
              for the next merger such that cost(first candidate, marked subnet)
              is minimized
            {comment:  a tie is first broken by weight and second by random
              selection};
            merge the two candidates into a single subnet
            {comment:  an unmarked subnet corresponding to a node at the
              next level is formed}
        end
    end
end
```

3.5 Time and Space Requirements

After a tree has been planted for a given network, the preprocessor calculates the time and space requirements of that tree (to compute specified performance measures). The time required to compute $G(\underline{N})$ is equal to the sum of the time required to compute g arrays for all the leaf nodes using either (9) or (A4) and the time requirements given by (8) for the M-1 convolutions. The space requirement for the computation of $G(\underline{N})$ is the maximum value of the sum of space requirements of g arrays that need to be saved by the algorithm at the same time. The number of g arrays that need to be saved at the same time depends upon the tree traversal order. For example, with postorder traversal of a balanced binary tree, the maximum number of arrays needed at the same time is $2 + \log_2 M$. Note that since partially covered arrays are of different sizes, the number of arrays needed does not

necessarily determine the space requirement. (For a detailed treatment of the accounting of time and space requirements, see Reference 13.)

Since space is reusable, the space needed to compute specified performance measures will be about the same as that for computing $G(\underline{N})$. However, the time needed to compute specified performance measures will be substantially more than the time to compute a single normalization constant. Tree traversals to compute performance measures efficiently and space-time tradeoffs are described in Section 4.

The time and space requirements of tree planting procedures (those investigated in [17, 18]) are very small compared to the requirements of array convolutions. For example, Algorithm 2 has a space requirement of $O(KM)$ and a time requirement of $O(KM^2)$. Also the operations required are mostly additions and comparisons rather than multiplications.

4. COMPUTATION OF PERFORMANCE MEASURES

Since all chains are fully covered at the root node of a tree, its g array degenerates to a single value, namely, the normalization constant $G(\underline{N})$. The computation of network performance measures, such as chain throughputs and mean queue lengths, requires the computation of various other normalization constants. (For a tutorial treatment of this topic, see [3] or [24].)

The throughput of chain k at center m for a network of closed chains with population vector $\underline{N}$ is [4, 7, 22]

$$T_{mk}(\underline{N}) = \lambda_{mk} \frac{G(\underline{N} - \underline{1}_k)}{G(\underline{N})} \text{ for } k = 1,2,\ldots,K,\ m = 1,2,\ldots,M \text{ and } \underline{N} \geq \underline{1}_k \qquad (11)$$

where $G(\underline{N} - \underline{1}_k)$ is the normalization constant of the same network with population vector $\underline{N} - \underline{1}_k$ and λ_{mk} is the relative arrival rate of chain k customers to center m. (11) is applicable for both fixed-rate and queue-dependent service centers.

The number of chain k customers in a service center (say m) is equal to zero if chain k is noncovered and is equal to N_k if chain k is fully covered by center m. To compute $q_{mk}(\underline{N})$, the mean number of chain k customers in center m, we need only to consider chains partially covered by center m. If center m is a fixed-rate service center, then the mean number of chain k customers in it is [22]

$$q_{mk}(\underline{N}) = \rho_{mk} \frac{G_{m+}(\underline{N} - \underline{1}_k)}{G(\underline{N})} \text{ for } k = 1,2,\ldots,K,\ m = 1,2,\ldots,M \text{ and } \underline{N} \geq \underline{1}_k \qquad (12)$$

where G_{m+} is the result of convolving p_m with

$$g_{\{1,2,\ldots,M\}} = p_1 \circledast p_2 \circledast \cdots \circledast p_M.$$

A queue-dependent service center with $\mu_m(i) = i$ is called an Infinite Server (IS) service center. The mean queue length of chain k here is [22]

$$q_{mk}(\underline{N}) = \rho_{mk} \frac{G(\underline{N} - \underline{1}_k)}{G(\underline{N})} = T_{mk}(\underline{N})\ \tau_{mk} \qquad (13)$$

which is available if the chain throughput has been obtained. We shall not consider this case separately any further.

If center m is a queue-dependent server with a general service rate function, $q_{mk}(\underline{N})$ needs to be calculated from the marginal distribution of queue lengths in center m given by

$$p_m(\underline{n}_m) = \frac{p_m(\underline{n}_m) G_{m-}(\underline{N} - \underline{n}_m)}{G(\underline{N})} \text{ for } m = 1,2,\ldots,M,\ \underline{0} \leq \underline{n}_m \leq \underline{N} \qquad (14)$$

where $p_m(\underline{n}_m)$ was given by (2) and G_{m-} is the g array of the subnet consisting of all service centers except center m. The quantities

$$G(\underline{N} - \underline{1}_k),\ G_{m+}(\underline{N} - \underline{1}_k) \text{ and } G_{m-}(\underline{N} - \underline{n}_m)$$

needed for (11), (12) and (14) respectively can be interpreted as the normalization constants of appropriately defined networks with trees such as those illustrated in Fig. 4 for M = 8.

$G(\underline{N} - \underline{1}_k)$ is simply the normalization constant of the original tree (i.e., queueing network) for the population vector $\underline{N} - \underline{1}_k$. $G_{m+}(\underline{N} - \underline{1}_k)$ is the normalization constant for the population vector $\underline{N} - \underline{1}_k$ computed from a tree in which center m appears twice at two leaf nodes. Note that a chain that is fully covered by center m in the original tree is fully covered by center m and its "clone" in the modified tree but only partially covered by either one. $G_{m-}(\underline{N} - \underline{n}_m)$ is computed from a tree that is the original tree with center m deleted. As a result, the set of chains partially covered by center m remains partially covered at the root node. Hence, G_{m-} is an array indexed over $i_k = 0,1,\ldots,N_k$ for all k partially covered by center m.

The computation of each of $G(\underline{N} - \underline{1}_k)$, $G_{m+}(\underline{N} - \underline{1}_k)$ and $G_{m-}(\underline{N} - \underline{n}_m)$ for $k = 1,2,\ldots,K$ and $m \in$ CENTERS(k) separately from traversing an entire tree requires approximately the same amount of time and space as $G(\underline{N})$. Thus the computation of chain throughputs and mean queue lengths can be done with (probably) no additional space, compared to that of $G(\underline{N})$, but with a time requirement up to $(M+1)K$ times that of $G(\underline{N})$. If additional space is available, then some or all of the g arrays from the computation of $G(\underline{N})$ can be saved, and the computation of the other normalization constants can be accomplished without traversing an entire tree. We found that some modest increase in space can give rise to very substantial savings in time. These considerations are addressed in Sections 4.1 to 4.4. An illustration of trading space for time is shown in Section 6.

It is convenient for us to assume for the moment that there is space to accommodate the entire tree of g arrays computed in the process of getting $G(\underline{N})$, in addition to temporary space needed for tree traversal and array convolutions. The time and space tradeoff of storing only some of the g arrays in the tree is addressed in Section 4.4 (see also Reference 13).

4.1 Marginal Distribution of Queue Lengths

If center m is a queue-dependent center, its mean queue lengths have to be calculated from the marginal queue length distribution of center m. We need the array G_{m-} first. Let σ_{pc} be the set of partially covered chains in center m. G_{m-} is an array indexed by $\underline{i}_{pc}$ and is obtained by redoing the convolutions along the path between center m and the root of the tree. We illustrate this with a binary tree in Fig. 5. For a balanced tree, the number of convolutions needed to get G_{m-} is $(\log_2 M)-1$. In Fig. 5, the sequence of convolutions needed is indicated by a dashed line. The stored g arrays needed at various nodes are labeled by g. Note that with a sequential convolution algorithm, G_{m-} is available free for m = M but requires M-1 convolutions to compute for m = 1,2,...,M-1.

4.2 Mean Queue Lengths For a Fixed-rate Service Center

To compute $q_{mk}(\underline{N})$, we need $G_{m+}(\underline{N} - \underline{1}_k)$, which is obtained by redoing the convolutions along the path from center m to the root of the tree. (See Fig. 6.) For a balanced tree, the number of convolutions needed is $(\log_2 M)+1$. The stored g arrays needed at various nodes are labeled by g in Fig. 6.

Let σ_{pc} be the set of partially covered chains in center m. Note that $G_{m+}(\underline{N} - \underline{1}_k)$ needs to be computed for every k in σ_{pc}. Some additional space will enable the computation of $G_{m+}(\underline{N} - \underline{1}_k)$ for all $k \varepsilon \sigma_{pc}$ to be performed at the same time. Instead of computing a single g array at a node along the path between center m and the root, multiple g arrays are computed. Recall that in the computation of $G(\underline{N})$, when a chain, say h, becomes fully covered at a node, the partially covered array computed for the node consists of elements with index value $i_h = N_h$. If both $G(\underline{N})$ and $G(\underline{N} - \underline{1}_h)$ are desired, then two partially covered arrays need to be computed at the node; one array contains elements with index value $i_h = N_h$ and the other contains elements with index value $i_h = N_h - 1$.

The method is best illustrated with an example. Let $\sigma_{pc} = \{1,2,3\}$ in the example. Referring to Fig. 6, suppose that chain 1 is fully covered at node 2, chain 2 is fully covered at node 3 and chain 3 is fully covered at node 4. Then, partially covered arrays needed for each node are shown in Table 6.

For a fixed-rate service center, the method just described to compute $G_{m+}(\underline{N} - \underline{1}_k)$, and thus mean queue lengths in center m, is likely to require less time and space than the computation of G_{m-} in the previous section. Two more convolutions are required in each tree traversal here. However, when a chain, say h, becomes fully covered, only array elements with index values N_h and N_h-1 are computed instead of elements for the full range of index values $\{0,1,\ldots,N_h\}$ needed in the computation of G_{m-}.

4.3 Chain Throughputs

We describe two methods for computing the normalization constants $G(\underline{N} - \underline{1}_k)$ for k = 1,2,...,K needed to calculate chain throughputs.

Method 1

Consider chain k which is partially covered by center m. Let NODE denote the (branch or root) node at which chain k becomes fully covered. An array convolution is performed at this node to obtain g array elements with index value $i_k = N_k - 1$. Convolutions at nodes along a path from NODE to the root node are then performed sequentially. The resulting normalization constant at the root node is $G(\underline{N} - \underline{1}_k)$. Consider the example illustrated in Fig. 7. Suppose that chain 1 visits centers 1, 14 and 16, and chain 2 visits centers 1, 3 and 7. Chain 1 does not become fully covered until the root node. Hence, one convolution (at the root node) is sufficient to compute $G(\underline{N} - \underline{1}_k)$ for k = 1. Chain 2 becomes fully covered at node 1. Two convolutions are thus needed, the first at node 1 and the second at the root node, to compute $G(\underline{N} - \underline{1}_k)$ for k = 2.

If chain k is fully covered by center m, then all convolutions along the path from center m to the root node need to be performed. The g array of center m, given by (9), can be obtained from the stored g array at the leaf node corresponding to center m as follows:

$$g_{\{m\}}(\underline{i}_{pc}) \leftarrow \frac{\mu(n_m)N_k}{n_m \rho_{mk}} g_{\{m\}}(\underline{i}_{pc}) \quad \text{for } i_h = 0,1,\ldots,N_h,\ h\varepsilon\sigma_{pc}. \tag{15}$$

Method 2

The normalization constants $G(\underline{N} - \underline{1}_k)$ for $k = 1,2,\ldots,K$ are computed together in the same tree traversal as $G(\underline{N})$; this is similar to the computation of $G_{m+}(\underline{N} - \underline{1}_k)$ described earlier. Let σ_{fc} denote the set of fully covered chains at some node in the tree and $\underline{i}_{fc} = \{i_k,\ k\varepsilon\sigma_{fc}\}$. At this node, $|\sigma_{fc}| + 1$ partially covered arrays are computed corresponding to the index values $\underline{i}_{fc} = \underline{N}_{fc}$ and $\underline{i}_{fc} = \underline{N}_{fc} - \underline{1}_k$ for $k\varepsilon\sigma_{fc}$, where $\underline{N}_{fc} = \{N_k,\ k\varepsilon\sigma_{fc}\}$. The partially covered arrays at the root node will then be equal to the normalization constants $G(\underline{N})$ and $G(\underline{N} - \underline{1}_k)$ for $k = 1,2,\ldots,K$.

4.4 Space-Time Tradeoff

In the discussions above on the computation of network performance measures, it was assumed for ease of exposition that the whole tree of g arrays from the computation of $G(\underline{N})$ was stored. It should be obvious from the methods described for the computation of the normalization constants $G(\underline{N} - \underline{1}_k)$ and $G_{m+}(\underline{N} - \underline{1}_k)$ and the array G_{m-} that g arrays at nodes near the root of the tree are fewer in number and are used much more frequently than g arrays at nodes near the leaves of the tree.

If space is limited so that only a few g arrays can be stored, then the g arrays at nodes immediately below the root node should be stored. In this case,

when g arrays not stored are required during a tree traversal, they are recomputed. An interesting optimization problem is as follows: given an amount of space available, which g arrays should be stored to minimize the time requirement to compute some specified performance measures? The numerical example in Section 6 shows that storing just the two g arrays of the root's sons enabled us to reduce the time requirement of computing chain throughputs very substantially.

Conceptually, we can think of partitioning the tree into subtrees such as shown in Fig. 8. The subtree of g arrays containing the root node (T_0 in Fig. 8) is saved and stored in memory. The g arrays in the other subtrees are not saved but are recomputed when needed. Alternatively, for very large queueing networks whose time requirements to recompute g arrays in these subtrees are very large, we can save them in secondary storage and swap them into memory when they are needed. The tree of g arrays thus provides a convenient data structure for implementing such storage management strategies to facilitate the solution of very large networks.

5. THE TREE ALGORITHM

All aspects of the tree convolution algorithm have been discussed in Sections 3 and 4. A high-level description of the entire algorithm is presented here.

Algorithm 3 {the tree algorithm}

```
begin
1.      input CENTERS(k), N_k, ρ_mk and μ_m for k = 1,2,...,K and m = 1,2,...,M,
          and performance measures desired by the analyst;
        repeat
2.          call a tree planting procedure;
3.          evaluate the time and space needed to calculate the specified
              performance measures
4.      until the analyst quits
5.      if a tree has been found with acceptable time and space requirements then
        begin
6.          determine which nodes of the tree of g arrays from the computation of
              G(N) should be saved
            {comment:  time-space tradeoff decision};
7.          postorder tree traversal to compute G(N)
            {comment:  when a node is visited, its g array is computed using Eq.
              (9) or Eq. (A4) for a leaf node, and Eq. (A1) for a branch node or
              the root node};
8.          tree traversals to compute those normalization constants G(N - 1_k),
              G_m+(N - 1_k) and G_m-(N - n_m) that are needed to evaluate the
              specified performance measures;
9.          output results
        end
    end
```

Algorithm 3 is made up of two procedures. Steps 1-6 constitute the preprocessor described in Section 3.4. Our current implementation of the preprocessor leaves some of the decisions for the programmer to make. First, the programmer specifies which tree planting procedure should be called in step 2. In steps 4 and 5, the programmer decides whether the time and space requirements of the tree algorithm are acceptable. (Are they better than those of other computational algorithms? Are they feasible for the computer being used?) Further, he may use a variety of tree planting procedures to plant several trees and then pick the best one. In step 6, the programmer decides whether some or all of the g arrays from the computation of $G(\underline{N})$ are saved for subsequent tree traversals (as described in Section 4.4).

Steps 7-9 constitute the second procedure that carries out the primary function of the tree algorithm, namely, preformance evaluation of a product-form queueing network. The details of step 7 have been given in Sections 3.1 and 3.2. The details of step 8 have been given in Sections 4.1 to 4.3.

6. A NUMERICAL EXAMPLE

We illustrate in this section the application of the tree algorithm to solve a queueing network model of the store-and-forward packet-switching network shown in Fig. 9. The network has 26 store-and-forward nodes and 64 communication channels (each link in Fig. 9 consists of 2 communication channels in opposite directions).

Since processor delays within store-and-forward nodes are typically much smaller than communication channel delays, they have been ignored in the queueing network model [15, 20]. The queueing network model thus has 64 queues with fixed-rate servers, one for each of the 64 communication channels.

The network supports 32 virtual channels with routes given in Table 7. Each virtual channel is modeled as a closed chain with the chain population size corresponding to the flow control window size of a virtual channel. For simplicity, we have ignored the modeling of end-to-end acknowledgements and the modeling of packet sources of virtual channels. (The interested reader is referred to [15, 20] for discussions on these modeling issues.) It is assumed that every packet (customer) arriving at its destination node triggers instantaneously the arrival of a new packet to the source node of the virtual channel. As a result, the number of packets within each virtual channel is fixed (closed chain model).

We provide solutions to the network example for two cases. In the first case, we consider all virtual channels to have a window size of 3, i.e., $N_k = 3$ for all k. It has been shown to be desirable to make the window size of a virtual channel equal to its path length, i.e., the number of communication channels along the route from the source node to the destination node [8, 14, 20]. This is the second case that we solved.

A communication channel in the network connecting node i to node j is named by the ordered pair (i,j) or $i \rightarrow j$. The mapping between the 64 communication channels and the 64 service centers in the model indexed by m = 1,2,...,64 is shown in Fig. 10, which also shows the tree planted by Algorithm 2. Each communication channel is modeled as a fixed-rate server with a mean service time of 1 second for all servers and all chains.

The performance measures of interest in the network example are the throughputs and mean end-to-end delays of the individual virtual channels. Using the tree algorithm, we computed $G(\underline{N})$ and $G(\underline{N} - \underline{1}_k)$ for k = 1,2,...,64 for the two cases of window sizes. The throughput of a virtual channel is computed using (11). The mean end-to-end delay of a virtual channel is then obtained from Little's formula. Results for the case of N_k equal to the path length of chain k for all k are shown in Table 8.

$G(\underline{N} - \underline{1}_k)$ was computed using two slightly different methods. In the first method, none of the g arrays from the computation of $G(\underline{N})$ was saved. An entire tree of g arrays is computed to get each $G(\underline{N} - \underline{1}_k)$. In the second method, the two g arrays at the root's sons from the computation of $G(\underline{N})$ are saved and stored. The time requirement of method 2 was found to be substantially less than that of method 1. The amount of space required is only slightly more. The actual number of multiplications, divisions and additions needed by the tree algorithm to obtain the results for the two cases were counted and are shown in Tables 9 and 10 for the two methods.

We now explain the space requirements given the use of static storage allocation or dynamic storage allocation shown in Tables 9 and 10. By dynamic allocation is meant that each g array is allocated storage for the exact number of array elements. By static allocation is meant that all g arrays are stored in data

structures of the same type (size). Furthermore, the type of the data structure is declared before the array convolutions are performed. In both cases, storage is allocated to an array only when needed. (The tree algorithm is currently implemented in the language Pascal with static allocation of storage for arrays.) With static allocation, the space requirement is the space needed by the maximum number of arrays that the algorithm needs to store at the same time. With dynamic allocation, the space requirement is the space needed to store the maximum number of array elements that the algorithm needs to store at the same time. After a tree has been planted, the preprocessor has sufficient information to calculate the space requirement given the use of either dynamic or static allocation.

For the network example considered, the preprocessor found that the maximum number of partially covered chains is 4 at any node in the tree of Fig. 10, and that each g array can be stored in a data structure with 4^4 elements in case 1 and with 7×6^3 elements in case 2. With method 1, the maximum number of g arrays that need to be stored in the postorder tree traversal, is $2 + (\log_2 M) = 8$. Hence, the space requirement given static allocation is $8 \times 4^4 = 2{,}048$ locations for case 1 and $8 \times 7 \times 6^3 = 12{,}096$ locations for case 2. With method 2, the two g arrays at the root's sons can be saved and $G(\underline{N} - \underline{1}_k)$ computed with additional storage for one more array. Hence, the space requirement given static allocation is $9 \times 4^4 = 2{,}304$ locations for case 1 and $9 \times 7 \times 6^4 = 13{,}608$ locations for case 2. Compare Table 9 and Table 10 and note that in this example a small increase in space buys a large amount of saving in time.

To calculate the chain throughputs for the network example using the sequential convolution algorithm and MVA algorithm, the space and time requirements for the case of $N_k = 3$ for all k are shown in Table 11. The results in Table 11 are based upon the original descriptions of the two algorithms and

Zahorjan's analysis of them [24]. (Other algorithm implementations may have slightly different time and space requirements. Their orders of magnitudes, however, are expected to be about the same. In particular, the MVA space requirement shown in Table 11 may be reduced by about a factor of K.) Note that with the MVA algorithm, mean queue lengths are also obtained for the same time requirement shown in Table 11. With the sequential or tree convolution algorithm, however, additional time is needed to compute mean queue lengths.

7. CONCLUSIONS

We have presented the tree convolution algorithm for the computation of normalization constants and performance measures of product-form queueing networks (Algorithm 3). The algorithm is very efficient, compared to existing algorithms, in the solution of networks with many queues and many routing chains that are characterized by a sparseness property. It is noted that the sparseness property and also a locality property are often encountered in models of large communication networks and distributed systems.

The tree algorithm exploits the routing information of a given network to reduce the computational time and space requirements of needed computations. The time and space savings are made possible by two features of the algorithm. First, the sequence of array convolutions to compute a normalization constant is determined by the traversal of a tree. Second, convolutions are performed between partially covered arrays that are much smaller, for networks with the sparseness property, than the K-dimensional arrays used by existing algorithms. The network routing information is utilized to configure the tree to reduce the time and space requirements of needed computations.

Algorithm 1 presents the basic algorithm of a family of heuristic procedures for tree planting. These procedures have been found to be very effective by an experimental study [17, 18] in which hundreds of networks were generated randomly and their computational time and space requirements determined. An exact solution of a communication network model with 64 queues and 32 routing chains is illustrated. Algorithm 2 presents the specific tree planting procedure used for the solution of this example. The large time and space savings of the tree algorithm in the example, compared to the requirements of the sequential convolution and MVA algorithms, are typical of models of large communication networks which are often characterized by strong sparseness and locality properties.

A tree is used because it is a convenient structure for representing an arbitrary sequence of array convolutions to compute normalization constants. Furthermore, a tree of arrays provides a flexible data structure for achieving space-time tradeoffs and for the incorporation of storage management techniques (to facilitate the solution of very large networks).

Given a network and its routing information, a preprocessor is used to construct trees and to evaluate the time and space needed to accomplish certain computations. The time and space requirements of the preprocessor itself are modest (much smaller than the requirements of array convolutions). The preprocessor provides fast accurate (a priori) estimates of the time and space needed to solve a specific network.

An analysis of the time and space complexity of the tree algorithm for a class of networks requires a model of the routing behavior of all networks in the class. In Reference 13, an analysis is presented for a class of networks whose routes are determined probabilistically by Bernoulli trials. The analysis

quantifies the expected time and space savings (as a function of a measure of sparseness) due to the use of partially covered arrays. Improvements due to tree optimization by tree planting procedures have been characterized experimentally [17, 18].

The sequential convolution algorithm, the MVA algorithm and their variants have time and space requirements that contain the term $\prod_{k=1}^{K} (N_k+1)$ which is the factor limiting the applicability of these algorithms. The limiting factor in the tree algorithm's time and space requirement is the maximum value of $\prod_{k\varepsilon\sigma_{pc}} (N_k+1)$ over all tree nodes, where σ_{pc} is the set of partially covered chains at a node. In general, if a tree can be found so that $|\sigma_{pc}| << K$ for each tree node, then the tree algorithm will provide substantial time and space savings. This is expected to be the case in the solution of large networks with the sparseness property. It should be obvious that the tree algorithm can solve a lot of networks that are not solvable by the sequential convolution and MVA algorithms. It should also be obvious that the tree algorithm cannot solve arbitrarily large networks. Therefore, the study of approximate solution techniques is still important. Since the tree algorithm provides an exact solution, approximate solution techniques can now be validated over a much larger set of product-form queueing networks than was previously possible without resorting to simulation.

Acknowledgements

The authors thank the editor Herb Schwetman and the anonymous reviewers for their constructive criticisms. They would also like to express their appreciation to the following people who provided helpful comments: Peter Denning and James Solberg of Purdue University, James C. Browne, K. Mani Chandy and A. Udaya Shankar of the University of Texas at Austin, Steve Lavenberg and Charles Sauer of IBM T. J. Watson Research Center, Paul Schweitzer of the University of Rochester, and John Zahorjan of the University of Washington.

REFERENCES

[1] Bard, Y., "Some Extensions to Multiclass Queueing Network Analysis," Proc. 4th International Symposium on Modelling and Performance Evaluation of Computer Systems, Vienna, Austria, Feb. 1979.

[2] Baskett, F., K. M. Chandy, R. R. Muntz, and F. Palacios, "Open, Closed and Mixed Networks of Queues with Different Classes of Customers," JACM, April 1975, pp. 248-260.

[3] Bruel, S. C. and G. Balbo, Computational Algorithms for Closed Queueing Networks, Elsevier, North-Holland, New York, 1980.

[4] Buzen, J. P., "Computational Algorithms for Closed Queueing Networks with Exponential Servers," Communications of the ACM, Sept. 1973, pp. 527-531.

[5] Chandy, K. M., U. Herzog and L. S. Woo, "Parametric Analysis of Queueing Networks," IBM J. of Res. and Develop., January 1975, pp. 43-49.

[6] Chandy, K. M. and D. Neuse, "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems," Comm. ACM, February 1982, pp. 126-134.

[7] Chandy, K. M. and C. H. Sauer, "Computational Algorithms for Product Form Queueing Networks," Comm. ACM, October 1980, pp. 537-583.

[8] Gerla, M. and L. Kleinrock, "Flow Control: A Comparative Survey," IEEE Trans. on Commun., April 1980, pp. 553-574.

[9] Kleinrock, L., Queueing Systems, Vol. 2: Computer Applications, Wiley-Interscience, New York, 1976, pp. 458-484.

[10] Lam, S. S., "Queueing Networks with Population Size Constraints," IBM J. of Research and Development, July 1977, pp. 370-378.

[11] Lam, S. S., "Dynamic Scaling and Growth Behavior of Queueing Network Normalization Constants," JACM, April 1982, pp. 492-513.

[12] Lam, S. S., "A Simple Derivation of the MVA and LBANC Algorithms from the Convolution Algorithm," Dept. of Computer Sciences, Univ. of Texas at Austin, Technical Report TR-184, November 1981.

[13] Lam, S. S. and Y. L. Lien, "An Analysis of the Tree Convolution Algorithm," Dept. of Computer Sciences, Univ. of Texas at Austin, Technical Report TR-166, February 1980.

[14] Lam, S. S. and Y. L. Lien, "Congestion Control of Packet Communication Networks by Input Buffer Limits--A Simulation Study," IEEE Trans. on Computers, October 1981, pp. 733-742.

[15] Lam, S. S. and J. W. Wong, "Queueing Network Models of Packet Switching Networks, Part 2: Networks with Population Size Constraints," Performance Evaluation, Vol. 2, No. 2, 1982 (to appear).

[16] Lavenberg, S., "Closed Multichain Product Form Queueing Networks with Large Population Sizes," Proc. of Interface between Applied Probability and Computer Science, Boca Raton, Florida, Jan. 1981.

[17] Lien, Y. L., "Modeling and Analysis of Flow-controlled Computer Communication Networks," Ph.D. Thesis, Dept. of Computer Sciences, University of Texas at Austin, December 1981.

[18] Lien, Y. L. and S. S. Lam, "The Design of Tree Planting Procedures for the Solution of Queueing Networks," Dept. of Computer Sciences, Univ. of Texas at Austin, Technical Report TR-180, 1982 (in preparation).

[19] Reiser, M., "Numerical Methods in Separable Queueing Networks," Studies in Management Sci., 7, (1977) pp. 113-142.

[20] Reiser, M., "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control," IEEE Trans. on Communications, August, 1979, pp. 1199-1209.

[21] Reiser, M., "Mean Value Analysis and Convolutional Method for Queue-dependent Servers in Closed Queueing Networks," Performance Evaluation, Vol. 1, No. 1, 1981, pp. 7-18.

[22] Reiser, M. and H. Kobayashi, "Queueing Networks with Multiple Closed Chains: Theory and Computational Algorithms," IBM J. of Research and Development, May 1975, pp. 283-294.

[23] Reiser, M. and S. S. Lavenberg, "Mean Value Analysis of Closed Multi-chain Queueing Networks," JACM, April 1980, pp. 313-322.

[24] Sauer, C. H. and K. M. Chandy, Computer Systems Performance Modeling, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[25] Schweitzer, P., "Approximate Analysis of Multiclass Closed Networks of Queues," International Conf. Stochastic Control and Optimization, Amsterdam, 1979.

[26] Wong, J. W. and S. S. Lam, "Queueing Network Models of Packet Switching Networks, Part 1: Open Networks," Performance Evaluation, Vol. 2, No. 1, 1982 (to appear).

[27] Zahorjan, J., "The Approximate Solution of Large Queueing Network Models," Ph.D. Thesis, available as Technical Report CSRG-122, Computer Systems Research Group, Univ. of Toronto, August 1980.

APPENDIX I

Derivation of time(SUBNET1, SUBNET2)

Equation (8) is evident when the convolution in (7) is rewritten in terms of elements of partially covered arrays. Let

$$\sigma_{10} = \{k_1, k_2, \ldots, k_a\} \subseteq \{1, 2, \ldots, K\}$$

$$\sigma_{11} = \{h_1, h_2, \ldots, h_b\} \subseteq \{1, 2, \ldots, K\}$$

and define

$$\sigma_x = \{k \mid \text{chain } k \text{ is partially covered by SUBNET1 and noncovered by SUBNET2}\}$$

and

$$\sigma_y = \{k \mid \text{chain } k \text{ is partially covered by SUBNET2 and noncovered by SUBNET1}\}$$

Then, (7) can be rewritten as

$$g_{SUBNET}(i_k, k\varepsilon\sigma_{01}U\sigma_{11})$$

$$= \sum_{j_{k_1}=0}^{N_{k_1}} \cdots \sum_{j_{k_a}=0}^{N_{k_a}} \sum_{j_{h_1}=0}^{i_{h_1}} \cdots \sum_{j_{h_b}=0}^{i_{h_b}} [g_{SUBNET1}(j_k, k\varepsilon\sigma_{10}U\sigma_{11}; i_k, k\varepsilon\sigma_x) \cdot g_{SUBNET2}(N_k - j_k, k\varepsilon\sigma_{10}; i_k - j_k, k\varepsilon\sigma_{11}; i_k, k\varepsilon\sigma_y)]$$

$$\text{for } i_k = 0, 1, \ldots, N_k, \; k\varepsilon\sigma_{01}U\sigma_{11} \qquad \text{(A1)}$$

APPENDIX II

Feedback filtering

Consider two leaf nodes {u} and {v}. Suppose that u is a fixed-rate service center. Let σ_v be the set of chains partially covered by {v} and σ_u be the set of chains partially or fully covered by {u}. Define

$$\underline{i}_{uv} = \{i_k,\ k\varepsilon\sigma_u \cup \sigma_v\}$$

Let $g_{\{u,v\}}$ be an array indexed by $\underline{i}_{uv}$. Define $g_{\{u,v\}}(\underline{0}) = 1$, where $\underline{0}$ is a zero vector of the appropriate dimension. Then, from (6) we get

$$g_{\{u,v\}}(\underline{i}_{uv}) = g_{\{v\}}(i_k, k\varepsilon\sigma_v)\ \delta(\underline{i}_{uv}) + \sum_{k\varepsilon\sigma_u} \rho_{uk}\ g_{\{u,v\}}(\underline{i}_{uv} - \underline{1}_k)$$

$$\text{for } i_k = 0,1,\ldots,N_k,\ k\varepsilon\sigma_u \cup \sigma_v \qquad \text{(A2)}$$

where $\underline{1}_k$ is a vector of the appropriate dimension with the component indexed by k equal to one and all other components equal to zero, $g_{\{u,v\}}(\underline{i}_{uv} - \underline{1}_k) = 0$ if $i_k = 0$, and

$$\delta(\underline{i}_{uv}) = \begin{cases} 0 \text{ if } i_k > 0 \text{ for any } k \notin \sigma_v \\ 1 \quad \text{otherwise} \end{cases}$$

The partially covered array for subnet {u,v} is then obtained from

$$g_{\{u,v\}}(\underline{i}_{pc}) = g_{\{u,v\}}(i_k, k\varepsilon\sigma_{pc}; N_k, k\varepsilon\sigma_{fc})$$

$$\text{for } i_k = 0,1,\ldots,N_k,\ k\varepsilon\sigma_{pc} \qquad \text{(A3)}$$

The array for {v} can also be obtained by feedback filtering if v is a fixed-rate service center. Redefine σ_v to be the set of chains partially or fully covered by {v}.

$$g_{\{v\}}(\underline{i}_v) = \sum_{k\epsilon\sigma_v} \rho_{vk}\, g_{\{v\}}(\underline{i}_v - \underline{1}_k)$$

$$\text{for } i_k = 0,1,\ldots,N_k,\ k\epsilon\sigma_v \qquad \text{(A4)}$$

where

$$\underline{i}_v = \{i_k,\ k\epsilon\sigma_v\}$$

$$g_{\{v\}}(\underline{0}) = 1$$

and

$$g_{\{v\}}(\underline{i}_v - \underline{1}_k) = 0 \text{ if } i_k = 0.$$

We can apply (A2) and (A3) to perform the convolution between a leaf node and its clone in mean queue length calculations discussed in Section 4. In this case, σ_u and σ_v in (A2) are the same and defined to be the set of chains partially or fully covered by the leaf node.

LIST OF TABLES

Chain k belongs to	Status of chain k: Overlapped by SUBNET1 and SUBNET2?	Status of chain k: Partially covered by SUBNET?
σ_{00}	no	no
σ_{01}	no	yes
σ_{10}	yes	no
σ_{11}	yes	yes

Table 1. Definition of the sets σ_{00}, σ_{01}, σ_{10} and σ_{11}.

	Traffic intensity ρ_{mk}			
	m = 1	m = 2	m = 3	m = 4
k = 1	0.5	1.0	0	0
k = 2	0.5	1.0	0.5	0
k = 3	0	0	0.5	1.0
k = 4	0	0	0.5	0

Table 2. Traffic intensities in the small example.

Subnets being merged	Overlapped chains	Chains fully covered after merger
{1}, {2}	1, 2	1
{3}, {4}	3	3, 4
{1, 2}, {3, 4}	2	1, 2, 3, 4

Table 3. Overlapped and fully covered chains in the small example.

(i_1, i_2)	$g_{\{1\}}(i_1, i_2)$
(0,0)	1
(0,1)	0.5
(1,0)	0.5
(1,1)	0.5
(2,0)	0.25
(0,2)	0.25
(1,2)	0.375
(2,1)	0.375
(2,2)	0.375

(i_1, i_2)	$g_{\{2\}}(i_1, i_2)$
(0,0)	1
(0,1)	1
(1,0)	1
(1,1)	2
(2,0)	1
(0,2)	1
(1,2)	3
(2,1)	3
(2,2)	6

(i_2, i_3)	$g_{\{3\}}(i_2, i_3)$
(0,0)	0.25
(0,1)	0.375
(1,0)	0.375
(1,1)	0.75
(2,0)	0.375
(0,2)	0.375
(1,2)	0.9375
(2,1)	0.9375
(2,2)	1.40625

i_3	$g_{\{4\}}(i_3)$
0	1
1	1
2	1

Table 4. g arrays at leaf nodes in the small example.

i_2	$g_{\{1,2\}}(i_2)$
0	$g_{\{1\}}(0,0)\ g_{\{2\}}(2,0) + g_{\{1\}}(1,0)\ g_{\{2\}}(1,0) + g_{\{1\}}(2,0)\ g_{\{2\}}(0,0)$ $= 1.75$
1	$g_{\{1\}}(0,0)\ g_{\{2\}}(2,1) + g_{\{1\}}(1,0)\ g_{\{2\}}(1,1) + g_{\{1\}}(2,0)\ g_{\{2\}}(0,1) +$ $g_{\{1\}}(0,1)\ g_{\{2\}}(2,0) + g_{\{1\}}(1,1)\ g_{\{2\}}(1,0) + g_{\{1\}}(2,1)\ g_{\{2\}}(0,0)$ $= 5.625$
2	$g_{\{1\}}(0,0)\ g_{\{2\}}(2,2) + g_{\{1\}}(1,0)\ g_{\{2\}}(1,2) + g_{\{1\}}(2,0)\ g_{\{2\}}(0,2) +$ $g_{\{1\}}(0,1)\ g_{\{2\}}(2,1) + g_{\{1\}}(1,1)\ g_{\{2\}}(1,1) + g_{\{1\}}(2,1)\ g_{\{2\}}(0,1) +$ $g_{\{1\}}(0,2)\ g_{\{2\}}(2,0) + g_{\{1\}}(1,2)\ g_{\{2\}}(1,0) + g_{\{1\}}(2,2)\ g_{\{2\}}(0,0)$ $= 11.625$

(a) Convolution to merge {1} and {2}.

i_2	$g_{\{3,4\}}(i_2)$
0	$g_{\{3\}}(0,0)\ g_{\{4\}}(2) + g_{\{3\}}(0,1)\ g_{\{4\}}(1) + g_{\{3\}}(0,2)\ g_{\{4\}}(0)$ $= 1.0$
1	$g_{\{3\}}(1,0)\ g_{\{4\}}(2) + g_{\{3\}}(1,1)\ g_{\{4\}}(1) + g_{\{3\}}(1,2)\ g_{\{4\}}(0)$ $= 2.0625$
2	$g_{\{3\}}(2,0)\ g_{\{4\}}(2) + g_{\{3\}}(2,1)\ g_{\{4\}}(1) + g_{\{3\}}(2,2)\ g_{\{4\}}(0)$ $= 2.71875$

(b) Convolution to merge {3} and {4}.

Table 5. g arrays at branch nodes in the small example.

Node	Status of chains 1, 2 and 3	Index values of fully covered chains in partially covered arrays
1	all partially covered	--
2	chain 1 fully covered	$i_1 = N_1, N_1-1$
3	chains 1 and 2 fully covered	$(i_1,i_2) = (N_1,N_2), (N_1-1,N_2), (N_1,N_2-1)$
4, Root	chains 1, 2 and 3 fully covered	$(i_1,i_2,i_3) = (N_1-1,N_2,N_3), (N_1,N_2-1,N_3), (N_1,N_2,N_3-1)$

Table 6. Arrays needed to obtain $G_{m+}(\underline{N} - \underline{1}_k)$ for k = 1,2,3 in example.

VIRTUAL CHANNEL	ROUTE (IN NODE SEQUENCE)
1	1 2 3 4 5 6 7
2	7 6 5 4 3 2 1
3	4 5 6 25
4	1 2 3 17 16
5	1 2 3 17 18
6	13 12 11 10 9 8
7	15 16 17 3
8	2 3
9	3 2
10	3 17 16 15
11	1 13 14 15 21 20
12	1 13 14 15 21 22
13	22 23 24 10 9
14	22 23 24 10 11
15	22 23 24 26
16	5 4 19
17	22 25 6 7 8
18	22 21 15 14 13 12
19	22 21 15 16 17 18
20	25 6 5 4
21	16 17 3 2 1
22	18 17 3 2 1
23	8 9 10 11 12 13
24	20 21 15 14 13 1
25	22 21 15 14 13 1
26	9 10 24 23 22
27	11 10 24 23 22
28	26 24 23 22
29	19 4 5
30	8 7 6 25 22
31	12 13 14 15 21 22
32	18 17 16 15 21 22

Table 7. Virtual channel routes of the store-and-forward network example.

CHAIN	THROUGHPUT RATE	DELAY
1	0.559	10.74
2	0.559	10.74
3	0.634	4.73
4	0.463	8.64
5	0.524	7.64
6	0.914	5.47
7	0.698	4.30
8	0.456	2.20
9	0.456	2.20
10	0.698	4.30
11	0.526	9.51
12	0.475	10.53
13	0.577	6.93
14	0.577	6.93
15	0.604	4.97
16	0.746	2.68
17	0.945	4.23
18	0.461	10.85
19	0.501	9.98
20	0.634	4.73
21	0.463	8.64
22	0.524	7.64
23	0.914	5.47
24	0.526	9.51
25	0.475	10.53
26	0.577	6.93
27	0.577	6.93
28	0.604	4.97
29	0.746	2.68
30	0.945	4.23
31	0.461	10.85
32	0.501	9.98

Table 8. Chain throughputs and mean end-to-end delays for the network example (N_k = path length of chain k for all k).

		Case 1 N_k = 3 for all k	Case 2 N_k = chain path length for all k
Time	multiplications	2,090,760	14,490,452
	divisions	154,962	545,946
	additions	1,935,534	13,944,372
Space	(if static allocation)	2,048	12,096
	(if dynamic allocation)	1,360	4,404

Table 9. Time and space requirements of the first method (g arrays not saved) for the network example.

		Case 1 N_k = 3 for all k	Case 2 N_k = chain path length for all k
Time	multiplications	991,132	6,751,230
	divisions	73,876	253,100
	additions	917,156	6,498,096
Space	(if static allocation)	2,304	13,608
	(if dynamic allocation)	1,376	4,476

Table 10. Time and space requirements of the second method (g arrays of root's sons saved) for the network example.

		The sequential convolution algorithm	The MVA algorithm
Time	multiplication	3.78×10^{22}	7.56×10^{22}
	divisions	32	5.90×10^{20}
	additions	3.78×10^{22}	7.61×10^{22}
Space		1.84×10^{19}	9.89×10^{21} (upper bound)

Table 11. Time and space requirements of the sequential convolution algorithm and the MVA algorithm for the network example (Case 1 $N_k = 3$ for all k).

LIST OF FIGURES

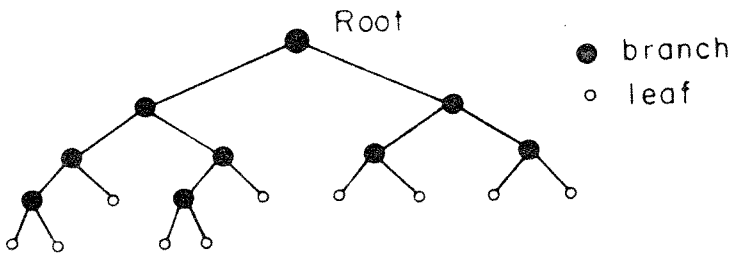

Fig. 1. A binary tree.

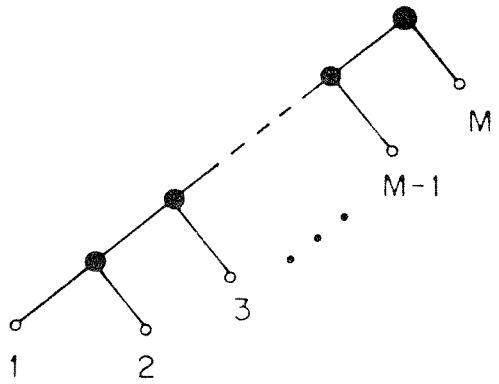

Fig. 2. Tree for the sequential convolution algorithm.

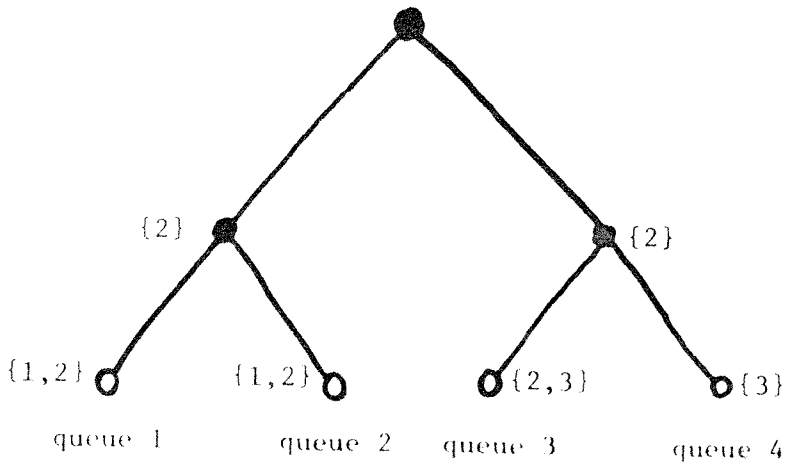

Fig. 3. Binary tree and partially covered chains in the small example.

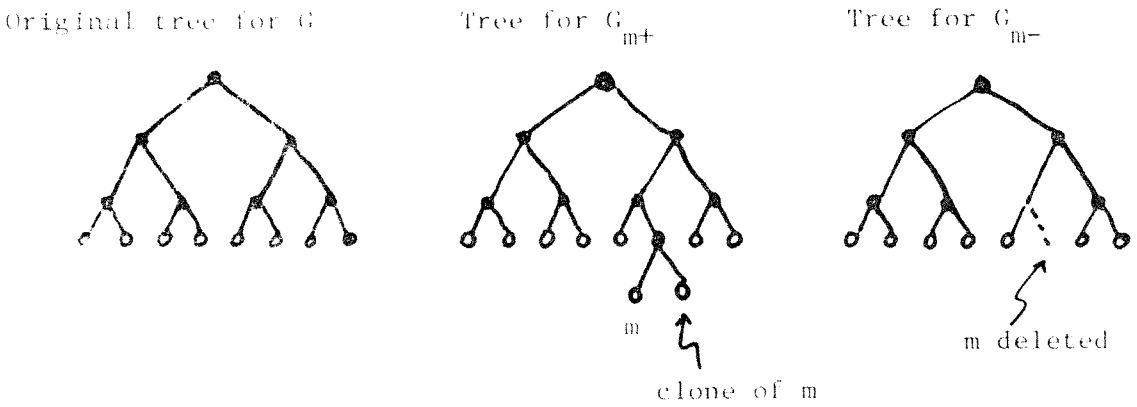

Fig. 4. Trees for G, G_{m+} and G_{m-}.

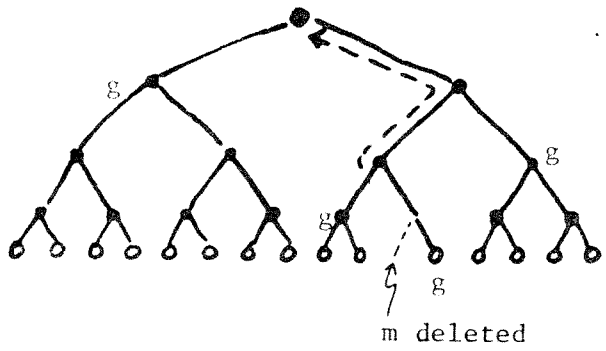

Fig. 5. Tree traversal to compute the array G_{m-}.

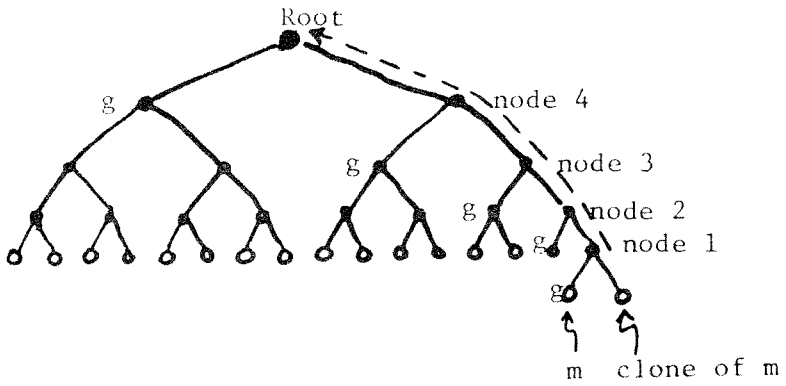

Fig. 6. Tree traversal to compute $G_{m+}(\underline{N} - \underline{1}_k)$.

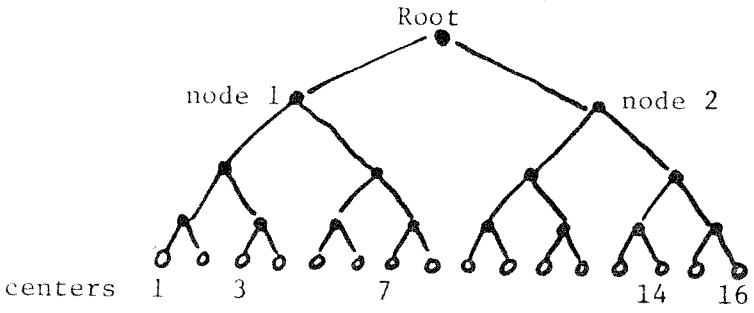

Fig. 7. Tree traversal to compute $G(\underline{N} - \underline{1}_k)$.

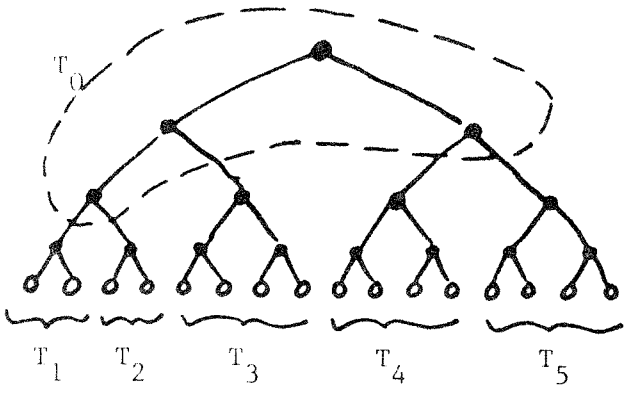

Fig. 8. An example of partitioning the tree of g arrays.

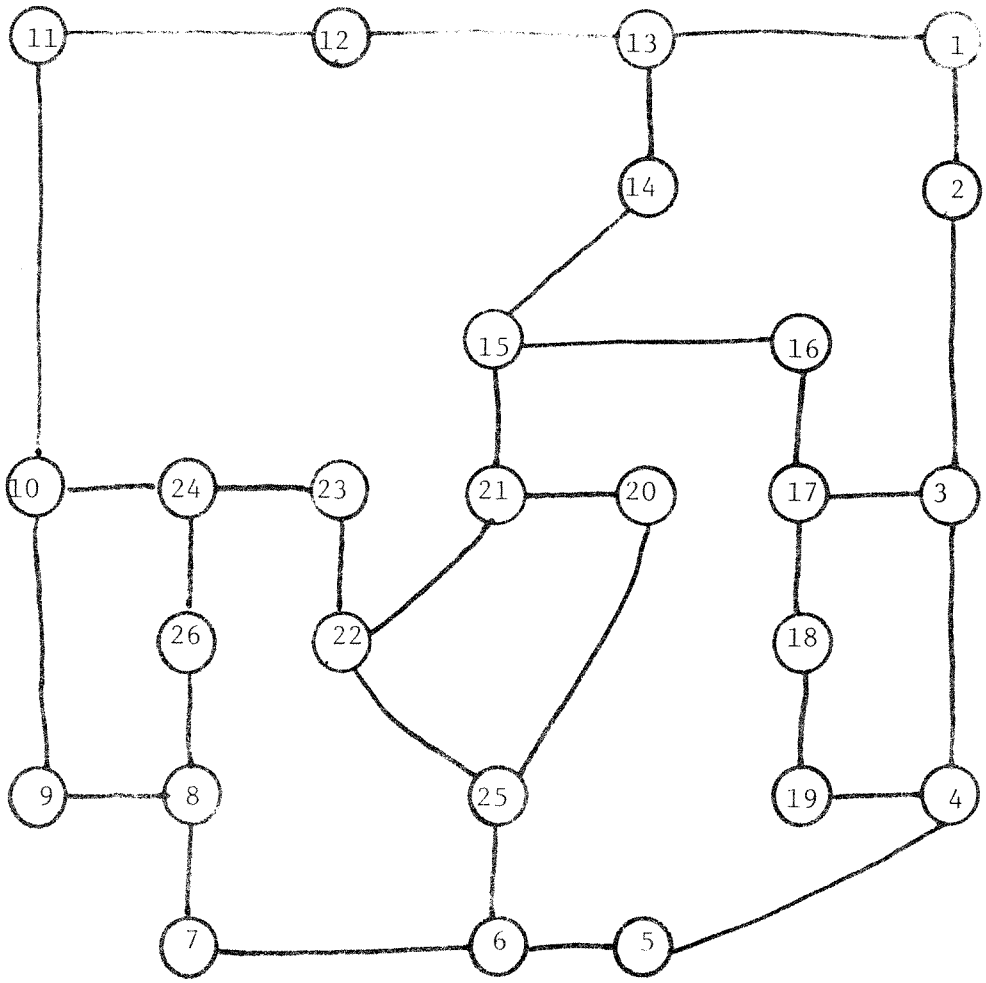

Fig. 9. Store-and-forward network of 26 nodes and 64 communication channels.

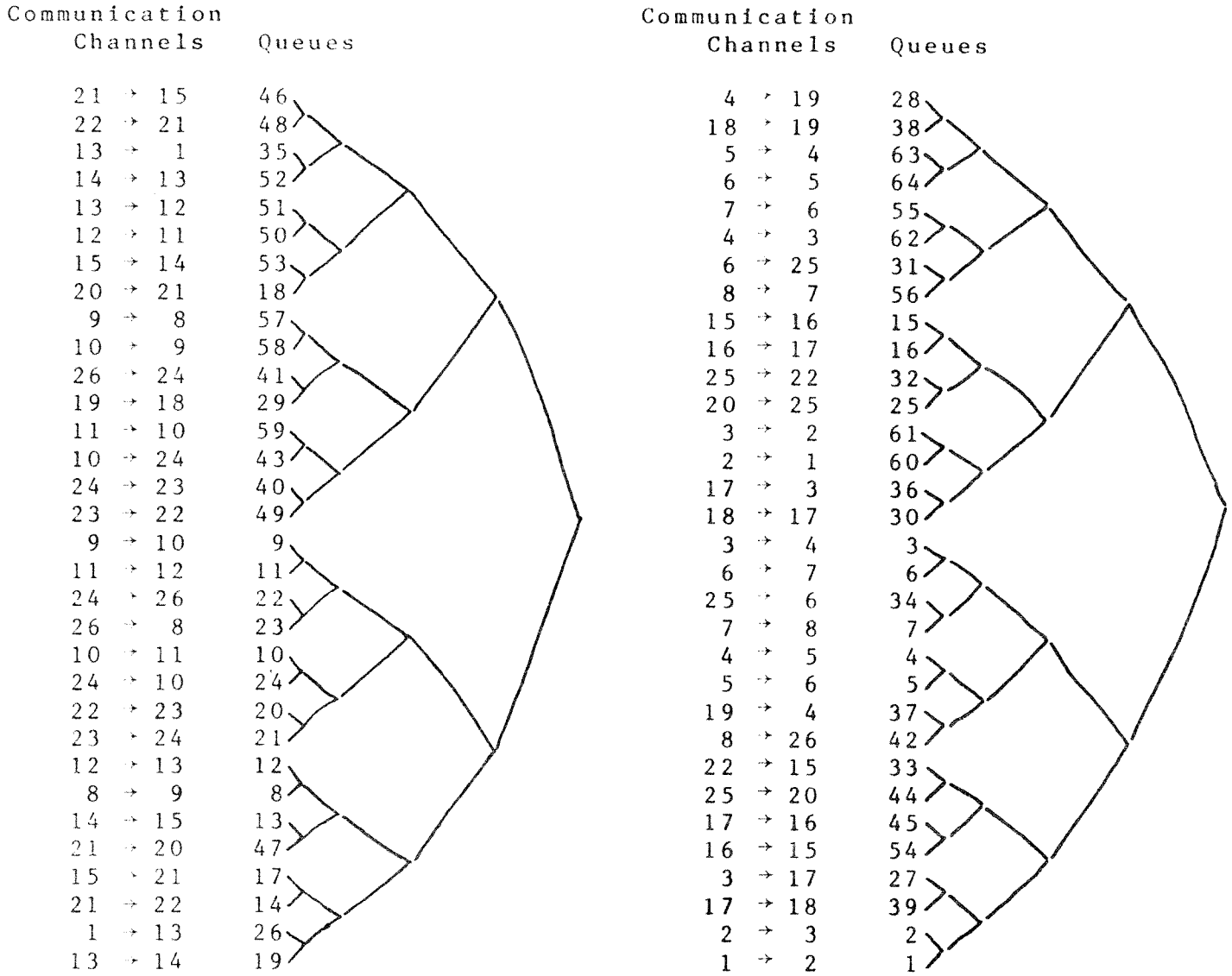

Fig. 10. Left and right subtrees planted for the network example.