Fast Algorithms for Generating Random Variates with Changing Distributions

SANGUTHEVAR RAJASEKARAN and KEITH W. ROSS
University of Pennsylvania

One of the most fundamental operations when simulating a stochastic discrete-event dynamic system is the generation of a nonuniform discrete random variate. The simplest form of this operation can be stated as follows: Generate a random variable X that is distributed over the integers $1, 2, \ldots, n$ such that $P(X = i) = a_i/(a_1 + \cdots + a_n)$, where the a_i's are fixed nonnegative numbers. The well-known "alias algorithm" is available to accomplish this task in $O(1)$ time. A more difficult problem is to generate variates for X when the a_i's are changing with time. We present three rejection-based algorithms for this task, and for each algorithm we characterize the performance in terms of acceptance probability and the expected effort to generate a variate. We show that, under fairly unrestrictive conditions, the long-run expected effort is $O(1)$. Applications to Markovian queuing networks are discussed. We also compare the three algorithms with competing schemes appearing in the literature.

Categories and Subject Descriptors: F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous; I.6.1 [**Simulation and Modeling**]: Simulation Theory; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*

General Terms: Algorithms

Additional Key Words and Phrases: Queuing networks, randomized algorithms, simulation

1. INTRODUCTION

The problem of generating a nonuniform discrete random variate is fundamental to the simulation of any discrete-event stochastic system. The simplest version of this problem is to generate a variate for a random variable X such that $P(X = i) = p_i$, $i = 1, \ldots, n$, given that the p_i's do not change with time. The well-known "alias algorithm" (e.g., see [2, pp. 158–160]) takes $O(n)$ preprocessing time, after which it can generate a variate in $O(1)$ time. In this paper we are interested in developing efficient algorithms for the case of the p_i's changing with time. Such a procedure is desirable in the simulation of

The work of K. W. Ross was partially supported by NSF grant DDM 5-22863.
Authors' addresses: S. Rajasekaran, Department of Computer and Information Science, and K. W. Ross, Department of Systems, University of Pennsylvania, Philadelphia, PA 19104.

multidimensional Markov processes, such as those used to model queuing and telephone networks.

A definition of the problem is as follows: Suppose that for each $s = 0, 1, 2, \ldots$ we are given n nonnegative numbers $a_1(s), \ldots, a_n(s)$. Let $X(s)$, $s = 0, 1, \ldots$, be a sequence of independent random variables such that

$$P(X(s) = i) = \frac{a_i(s)}{a_1(s) + \cdots + a_n(s)}, \qquad i = 1, \ldots, n.$$

The problem is to generate a variate for $X(s)$ for each $s = 0, 1, \ldots$ (i.e., generate a realization of $\{X(s)\}$). We refer to the integers in $\{1, \ldots, n\}$ as the *outcomes* of X. Call $a_i(s)$ the rate for outcome i at time s. In the static version of the problem, the rates do not change with time. In the dynamic version, zero or more of the $a_i(s)$'s change each time s is incremented.

In discrete-event simulation, the number of $a_i(s)$'s that change each time s is incremented is typically small. The algorithms developed in this paper attempt to exploit this special property.

In Section 2 we first show how the rejection algorithm can be employed to generate variates with changing distributions. The performance of the algorithm can be characterized by its acceptance probability. We then develop two additional algorithms that, at the expense of additional memory, can improve significantly the acceptance probability. In Section 3 we suppose that the rates are random and are determined by a (discrete state space) Markov process, as is typically the case in applications. We explicitly characterize the long-run acceptance probability and the long-run expected effort of the three algorithms in terms of the steady-state rates of the underlying Markov process. In Section 4 several classes of queuing networks are considered, and for each class we characterize the long-run performance of the algorithms in terms of the defining parameters of the network. In Section 5 we compare the algorithms of this paper with the event-list approach, assuming that the event list is organized as a heap, and with the algorithm TRANSIT proposed by Fox [7]. In Section 6 we discuss several methods for improving the performance of the three algorithms discussed in this paper.

2. THREE ALGORITHMS

The following assumptions will be in force throughout this paper:

(A1) For each $i = 1, \ldots, n$, there exists a finite $\bar{a}_i$ such that $a_i(s) \le \bar{a}_i$ for all $s = 0, 1, 2, \ldots$. Denote $\bar{a} := \max\{\bar{a}_1, \ldots, \bar{a}_n\}$.

(A2) There exists a finite $b > 0$ such that

$$\sum_{i=1}^{n} a_i(s) \ge b \qquad \text{for all} \quad s = 0, 1, 2, \ldots.$$

We refer to $(\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_n)$ as the majorizing n-vector. All of the three algorithms presented in this paper are rejection based. Any rejection algorithm that operates with respect to a majorizing n-vector runs in $O(1)$

average time, provided the *acceptance probabilities* are bounded away from zero as n increases. Traditionally and correctly, however, a major theme in the literature on rejection algorithms is making the implicit constant (depending on the reciprocal of the acceptance probability) small. We quantify this constant in terms of the $\bar{a}_i$'s. In particular, we quantify this constant for product form instances of the queuing-network models in terms of the traffic conditions. Clearly, though, the heavier the traffic at all nodes, the smaller will be the constant for essentially any network.

We call our three algorithms Generate1, Generate2, and Generate 3. Generate1 is an implementation of a discrete version of standard rejection using the dynamic rate vector $(a_1(s), a_2(s), \ldots, a_n(s))$ with respect to the uniform majorizing n-vector $(\bar{a}, \bar{a}, \ldots, \bar{a})$. Generate3 is an implementation of a discrete version of standard rejection using the dynamic rate vector with respect to the nonuniform majorizing n-vector $(\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_n)$. Both Generate1 and Generate3 are based on assigning a single "bucket" for each event. Generate2 combines the principle in Generate1 with assignment of (possibly) more than one bucket to any outcome. Bucket schemes have been considered before in the literature, but apparently not linked directly to rejection algorithms.

2.1 Algorithm Generate1

We now present our first algorithm for generating variates with a changing distribution. Our algorithm makes use of n buckets, that is, an array B of size n. Each bucket has a "capacity" of 1. For each bucket i, we initially set $B[i] = a_i(0)/\bar{a}$. This completes preprocessing. Clearly, this processing can be completed in $O(n)$ time. Notice that the initial contents of each bucket does not exceed unity. Here is an algorithm to generate a variate for $X(0)$ (hereafter, let U stand for the uniform random variable in the interval $(0, 1)$):

Algorithm Generate1
Step 1. Generate U, and compute $I = \lfloor nU \rfloor$.
Step 2. Let $R = I - nU$.
Step 3. If $R \leq B[I]$, accept and output I and quit;
Else go to Step 1.

Observe that Generate1 picks a bucket at random (uniformly). If I is the bucket chosen, then I is output with probability $B[I]$. (Notice that R is a uniform variable in the interval $(0, 1)$ and is independent of I.)

The algorithm Generate1 is the rejection algorithm (e.g., see [2, pp. 151–152]) for generating discrete random variates with fixed distributions. (The rejection algorithm is typically discussed in the context of continuous random variables.) It follows that the output of the algorithm Generate1 has the same distribution as that of $X(0)$.

The key observation we make here is that Generate1 can also be used to generate variates with changing distributions. Indeed, suppose Generate1 is used to generate a variate for $X(s)$. (Thus, $B[i] = a_i(s)/\bar{a}$.) If at time $s + 1$ we have $a_i(s + 1) \neq a_i(s)$ for some i, then we simply reset $B[i] = a_i(s + 1)/\bar{a}$. The procedure is valid since, by Assumption (A1), the contents of each bucket never exceeds unity.

In order to characterize the performance of Generate1, let "acceptance at time s" be the event that $R \le B[I]$ the first time Step 3 is invoked when generating a variate for $X(s)$. We refer to P ("acceptance at time s") as the *acceptance probability at time s*. Note that the expected number of uniform random variates U needed to generate a variate for $X(s)$ by the algorithm Generate1 is equal to the reciprocal of the acceptance probability.

THEOREM 2.1 *The acceptance probability at time s for Generate1 is given by*

$$P(\text{"acceptance at time } s\text{"}) = \frac{\sum_{i=1}^{n} a_i(s)}{n\bar{a}}. \tag{1}$$

, PROOF. We have

$$\begin{aligned} P(\text{"acceptance at time } s\text{"}) &= P(R \le B[I]) \\ &= \sum_{i=1}^{n} \frac{1}{n} P(R \le B[i]) \\ &= \sum_{i=1}^{n} \frac{1}{n} \frac{a_i(s)}{\bar{a}}, \end{aligned}$$

establishing the result. □

From Theorem 2.1 we know that the performance of Generate1 will be poor at time s when the average of the $a_i(s)$'s is significantly less than the uniform upper bound $\bar{a}$. In particular, Generate1 does not exploit the fact that we may have $\bar{a}_i \ll \bar{a}$ for many outcomes i, as is often the case in applications. The algorithms discussed below, Generate2 and Generate3, take advantage of this special structure.

2.2 Algorithm Generate2

In Generate1 one bucket is assigned to each outcome i, $i = 1, \ldots, n$. The idea behind Generate2 is to assign one or more buckets to any outcome i. The number of buckets assigned to i is proportional to the upper bound of the ith rate. Specifically, let d be a fixed positive number, and let the number of buckets assigned to outcome i, denoted by l_i, be given by

$$l_i = \left\lceil \frac{\bar{a}_i}{d} \right\rceil, \qquad i = 1, \ldots, n.$$

Also, let $l = l_1 + \cdots + l_n$ be the total number of buckets. Note that we have the following bounds on the total number of buckets:

$$\frac{\sum_{i=1}^{n} \bar{a}_i}{d} \le l < \frac{\sum_{i=1}^{n} \bar{a}_i}{d} + n.$$

Thus, if $d = (\sum_{i=1}^{n} \bar{a}_i)/n$, then $n \le l \le 2n$.

In Generate2 the number of buckets assigned to any i is held fixed even when the $a_i(s)$'s change. Once having chosen at random one of the l buckets,

if that bucket is assigned to i, then we will output i with a certain probability. This probability, however, will change as the $a_i(s)$'s change.

To be more precise, we make use of two arrays C and B of size l and n, respectively. For B we set

$$B[i] = \frac{a_i(0)}{dl_i}, \qquad i = 1, \ldots, n.$$

We set $C[j]$ equal to the outcome i assigned to bucket j (for $1 \le j \le l$).

Preprocessing

Step 1. For each outcome i, compute l_i.
Step 2. Compute the prefix sums of $(l_1, l_2, \ldots, l_n)$. Let the sums of $(m_1, m_2, \ldots, m_n)$; that is, $m_i = \sum_{j=1}^{i} l_j$, for $i = 1, 2, \ldots, n$.
Step 3. Initialize C as follows: Fill cells 1 through m_1 with 1, cells $m_1 + 1$ through m_2 with 2, and so on.
Step 4. Set $B[i] = (a_i(0))/(dl_i)$ for $1 \le i \le n$.

Next we present the procedure for generating a variate for $X(0)$.

Algorithm Generate2

Step 1. Generate U, and compute $J = \lfloor lU \rfloor$.
Step 2. Let $R = J - lU$.
Step 3. Let $I = C[J]$. If $R \le B[I]$, output I and quit;
Else go to Step 1.

As with Generate1, this algorithm can also be used to generate a variate for $X(s)$. After having generated a variate for $X(s)$, if at time $s + 1$ we have $a_i(s + 1) \ne a_i(s)$ for some i, we simply reset $B[i] = a_i(s + 1)/dl_i$. Note that, since $a_i(s) \le \bar{a}_i$, $B[i]$ never exceeds unity. Also note that we can always set $\bar{a}_i = \bar{a}$, $i = 1, \ldots, n$, and $d = \bar{a}$, in which case Generate2 reduces to Generate1. In a manner entirely analogous to that for Generate1, we can define the event "acceptance at time s" for the algorithm Generate2.

THEOREM 2.2 *At time s, the output of Generate2 has the same distribution as that of $X(s)$. Moreover, for Generate2 we have*

$$P(\text{"acceptance at time } s\text{"}) = \frac{\sum_{i=1}^{n} (a_i(s)/d)}{\sum_{i=1}^{n} \lceil \bar{a}_i/d \rceil}. \tag{2}$$

PROOF. We have

$$P(\text{"acceptance at time } s\text{"}) = \sum_{i=1}^{n} \frac{l_i}{l} B[i],$$

from which (2) follows. Also, note that

$$P(I = i, \text{"acceptance at time } s\text{"}) = \frac{l_i}{l} B[i].$$

Combining the above two expressions gives $P(I = i|\text{"acceptance at time } s\text{"}) = a_i(s)/(a_1(s) + \cdots + a_n(s))$, which completes the proof. □

Theorem 2.2 directly gives the following result:

COROLLARY 2.1 *We have the following bounds on the acceptance probability for Generate2:*

$$\frac{\sum_{i=1}^{n} a_i(s)}{\sum_{i=1}^{n} \bar{a}_i + nd} < P(\text{"acceptance at time } s\text{"}) \leq \frac{\sum_{i=1}^{n} a_i(s)}{\sum_{i=1}^{n} \bar{a}_i}.$$

Furthermore, if d is a divisor of $\bar{a}_i$, $i = 1, \ldots, n$, then P("acceptance at time s") $= (\sum_{i=1}^{n} a_i(s))/(\sum_{i=1}^{n} \bar{a}_i)$.

Since $\sum_{i=1}^{n} \bar{a}_i$ can be significantly less than $n\bar{a}$, the algorithm Generate2 can give a much higher acceptance probability as compared with Generate1.

2.3 Algorithm Generate3

Our last algorithm is similar in spirit to Generate2. It has the advantage of always achieving the upper bound in Corollary 2.1 for the acceptance probability. It has the disadvantage that the effort required to pass through one iteration of the algorithm is somewhat greater than that required by Generate2. Generate3 makes use of the alias algorithm and its data structures. It also makes use of an array B of size n. Initially, we set $B[i] = a_i(0)/\bar{a}_i$. The preprocessing, including that required by the alias algorithm, is $O(n)$. An equivalent version of this algorithm has been independently discovered by Fishman [5].

Algorithm Generate3
Step 1. Use the alias algorithm to generate a variate I with distribution $q_i = \bar{a}_i/(\bar{a}_1 + \cdots + \bar{a}_n)$, $i = 1, \ldots, n$.
Step 2. Generate a uniform variate U.
Step 3. If $U \leq B[I]$, accept and output I and quit;
Else go to Step 1.

We leave it to the reader to establish the following analog to Theorem 2.2.

THEOREM 2.3 *At time s, the output of Generate3 has the same distribution as that of $X(s)$. Moreover, for Generate 3 we have*

$$P(\text{"acceptance at time } s\text{"}) = \frac{\sum_{i=1}^{n} a_i(s)}{\sum_{i=1}^{n} \bar{a}_i}. \tag{3}$$

We now give a brief comparison of the three algorithms. Throughout this comparison, suppose that $d = (\sum_{i=1}^{n} \bar{a}_i)/n$ in Generate2; hence, from Corollary 2.1, we know that the acceptance probability for Generate2 is greater than $\frac{1}{2}$ of the acceptance probability for Generate3. Table I gives a comparison of the memory requirements and the operation counts for the three algorithms. We have assumed the worst-case memory count for Generate2. For the operation counts, we have assumed that each of the following operations counts as 1: multiplication, subtraction, upper floor operation, table lookup, comparison, and uniform variate generation. We define the operation count as the maximum number of operations performed in one pass

Table I. Comparison of Three Algorithms

	Memory	Operation count
Generate1	n reals	6
Generate2	n reals, $2n$ integers	7
Generate3	$2n$ reals, n integers	10

through the steps of the algorithm. In particular, if the acceptance probability of Generate2 is close to that of Generate3, we may conclude from Table I that Generate2 is preferable to Generate3.

Also, if d is a divisor of each rate and if the rates do not change with time, Generate2 can be streamlined because $B[i] = 1$ for all i. Moreover, this streamlined version of Generate2 will run faster than the alias algorithm. The streamlined algorithm is similar, but not identical, to the algorithm developed in [2, pp. 157–158]. (Fox's algorithm has more flexibility due to its choice of parameter m.)

2.4 A Performance Measure for Randomized Algorithms

From Theorem 2.3 we know that the expected number of U's needed to generate a variate for $X(s)$ is not greater than $\sum_{i=1}^{n} \bar{a}_i/b$. However, Theorem 2.3 does not exclude the possibility that the number of U's needed to generate $X(s)$ exceeds $\sum_{i=1}^{n} \bar{a}_i/b$ with significant probability. Analogous statements can be made for Theorems 2.1 and 2.2.

In order to study this further, let $V = \sum_{i=1}^{m} \eta_i$, where η_i's are independent, identically distributed geometric random variables with parameter p. (V can be thought of as the number of times a coin has to be flipped before a head appears for the mth time, p being the probability that a head appears in a single flip). Chernoff bounds (see, e.g., [11, pp. 388–393]) can be used to obtain tight bounds on probabilities in the tail ends of V. In particular, we can show the following:

$$P\left[V \geq (1+\epsilon)\frac{m}{p}\right] \leq \exp\left(\frac{-\epsilon^2 m}{1-p}\right),$$

for any fixed $0 < \epsilon \ll 1$ (see [14]). We make use of these bounds in our analysis.

Just like the $O(\ )$ function is used to specify the asymptotic resource bounds of deterministic algorithms, $\tilde{O}(\ )$ is used to specify resource (like time and space) bounds of randomized algorithms. $\tilde{O}(\ )$ is defined as follows:

Definition [14]. We say that a function $f(\cdot)$ is $\tilde{O}(g(\cdot))$ if there exist constants c and n_0 such that $f(n) \leq c\alpha g(n)$ with probability $\geq (1 - n^{-\alpha})$ on any input of size $n \geq n_0$, for any $\alpha > 0$.

Definition. We say that a function $f(\cdot)$ is $\omega(g(\cdot))$ if $\lim_{n\to\infty}(g(n))/(f(n)) = 0$.

THEOREM 2.4 *Let h be a lower bound on the acceptance probability of any of the three algorithms. If this algorithm is called m times, then the total number of U's generated by this algorithm is no more than $(1 + \epsilon)(m/h)$ with probability $\geq 1 - e^{[-\epsilon^2 m/(1-h)]}$ for any fixed $0 < \epsilon \ll 1$.*

PROOF. Follows from the Chernoff bounds and the observation that the number of U's generated by any of the three algorithms to obtain a variate for X is upper bounded by a geometric random variable with parameter h. □

COROLLARY 2.2 *If $m = \omega(\log n)$, the number of U's generated is no more than $(1 + \epsilon)(m/h)$ with probability $\geq (1 - (1/n^{\alpha}))$ for any constant $0 < \epsilon \ll 1$ and any fixed $\alpha \geq 1$. Thus, the number of U's generated is $\tilde{O}(m/h)$.*

3. RANDOM RATES

In discrete-event simulation, the rates typically evolve in a random manner. In this context we shall write $A_i(s)$ for the ith rate at time s in order to emphasize the fact that the rates are random. Also, write $\mathbf{A}(s) = (A_1(s), \ldots, A_n(s))$ and $Y(s) := A_1(s) + \cdots + A_n(s)$ for each s. In the context of random rates, we no longer require that the variables $X(s)$, $s = 0, 1, \ldots$, be independent. We do require, however, that $X(s + 1)$ be independent of $\{X(0), \ldots, X(s)\}$ given $\mathbf{A}(s)$. It is easily seen that Generate1, Generate2, and Generate3 satisfy this requirement.

Assumptions (A1) and (A2) translate directly to the current context with $a_i(s)$ replaced by $A_i(s)$. The acceptance probabilities in Theorems 2.1, 2.2, and 2.3 have their analogs in the context of random rates: Replace $a_i(s)$ by $E[A_i(s)]$.

In most applications (see Section 4), there is an underlying (discrete state space) Markov process that defines the rates $A_1(s), \ldots, A_n(s)$. In this case $Y(s)$ is the total jump rate of the Markov process just before the sth jump. It is convenient to make the following assumption:

(A3) The underlying Markov process is irreducible, aperiodic, and positive recurrent.

With Assumption (A3) in force, $Y(s)$ will converge in distribution to a random variable Y that is the total rate of the process just before a jump in steady state. Let Z be the total rate of the process at an arbitrary time instant in steady state (which is also well defined due to Assumption (A3)). The random variables Y and Z take on the same discrete values, but typically have different distributions. In fact, it follows from [15, Theorem 2.10.6a] that

$$P(Y = y) = \frac{y}{E[Z]} P(Z = y) \qquad \text{for all} \quad y. \tag{4}$$

The following theorems enable us to characterize the long-run performance of the algorithms in terms of the steady-state behavior of the underlying Markov process. (Of course, not all simulations are steady state.) For each of the three algorithms, let $V(s)$ be the number of random variables U that are called in order to generate a variate for $X(s)$. We shall refer to $V(s)$ as the *effort* required to generate a variate for $X(s)$.

THEOREM 3.1 *Under assumptions (A1)–(A3), Generate1 has the following long-run acceptance probability and long-run expected effort:*

$$\lim_{s\to\infty} P(\text{"acceptance at time } s\text{"}) = \frac{E[Z^2]}{n\bar{a}E[Z]},$$

$$\lim_{s\to\infty} E[V(s)] = \frac{n\bar{a}}{E[Z]}.$$

PROOF. From Theorem 2.1 we have

$$P(\text{"acceptance at time } s\text{"}|\mathbf{A}(s)) = \frac{Y(s)}{n\bar{a}}$$

and

$$E[V(s)|\mathbf{A}(s)] = \frac{n\bar{a}}{Y(s)}.$$

Taking expectations and applying Assumption (A3) to both of these equalities give

$$\lim_{s\to\infty} P(\text{"acceptance at time } s\text{"}) = \frac{E[Y]}{n\bar{a}} \tag{5}$$

and

$$\lim_{s\to\infty} E[V(s)] = E\left[\frac{n\bar{a}}{Y}\right]. \tag{6}$$

But from (4) we have $E[Y] = E[Z^2]/E[Z]$ and $E[1/Y] = 1/E[Z]$, which, when combined with (5) and (6), give the desired result. □

The proofs of the following two theorems are entirely analogous to the proof of Theorem 3.1:

THEOREM 3.2 *Under assumptions (A1)–(A3), Generate2 has the following long-run acceptance probability and long-run expected effort:*

$$\lim_{s\to\infty} P(\text{"acceptance at time } s\text{"}) = \frac{E[Z^2]}{E[Z]d\Sigma_{i=1}^{n}\lceil \bar{a}_i/d\rceil},$$

$$\lim_{s\to\infty} E[V(s)] = \frac{d\Sigma_{i=1}^{n}\lceil \bar{a}_i/d\rceil}{E[Z]}.$$

THEOREM 3.3 *Under assumptions (A1)–(A3), Generate3 has the following long-run acceptance probability and long-run expected effort:*

$$\lim_{s\to\infty} P(\text{"acceptance at time } s\text{"}) = \frac{E[Z^2]}{E[Z](\Sigma_{i=1}^{n}\bar{a}_i)},$$

$$\lim_{s\to\infty} E[V(s)] = \frac{\Sigma_{i=1}^{n}\bar{a}_i}{E[Z]}.$$

4. APPLICATION TO QUEUING NETWORKS

In this section we give examples for which it is possible to calculate explicitly the moments appearing in the theorems of Section 3. Although some of these examples are better studied by analytical methods than by simulation, the results give insight into the run times of the Generate algorithms.

Queuing networks are used extensively in the modeling and analysis of computer systems and networks (e.g., see [10]). To illustrate our ideas for variate generation, consider the open Jackson network (e.g., see [15]), one of the most fundamental queuing networks. There are n single-server queues. Customers arrive from the outside according to a Poisson process with rate λ and are routed to queue i with probability r_{0i}, $i = 1, \ldots, n$. Service times at queue i are exponentially distributed with parameter μ_i, $i = 1, \ldots, n$. When a customer completes service at queue i, it is routed to queue j with probability r_{ij}; it leaves the network with probability $r_{i0} := 1 - \sum_{j \neq i} r_{ij}$. Service times are assumed to be independent of each other and of the arrival process.

Assume that there exists a nonnegative solution $(\lambda_1, \ldots, \lambda_n)$ to the "traffic equations,"

$$\lambda_i = \lambda r_{0i} + \sum_{j=1}^{n} \lambda_j r_{ji}, \qquad i = 1, \ldots, n,$$

such that $\rho_i := \lambda_i/\mu_i < 1$ for all $i = 1, \ldots, n$. This assumption assures that the underlying Markov process is irreducible, aperiodic, and positive recurrent.

Now consider simulating the Markov process associated with this queuing network. Note that the rate associated with queue i, $A_i(s)$, is either μ_i or 0 depending on whether queue i is occupied or empty. Note that the rate associated with external arrivals, $A_0(s)$, is λ. Given that the current rates are $A_0(s), \ldots, A_n(s)$, the elapsed time (the interevent time) until the next arrival or service-completion event is exponentially distributed with parameter $Y(s) := A_0(s) + \cdots + A_n(s)$. This event is a service completion at queue i with probability $A_i(s)/Y(s)$, $i = 1, \ldots, n$, and an arrival with probability $A_0(s)/Y(s)$. After having determined the elapsed time and event type, the alias algorithm can be used to determine, in $O(1)$ time, the queue to which the customer is to be routed (including the possibility of being routed to the outside). A new iteration then begins (i.e., s is incremented), during which we determine the next interevent time, the next event type, and the next queue to which the customer is routed. Note that at most two of the rates change from iteration to iteration.

Consider the central component of the above simulation procedure: determining the event type according to the probabilities $A_i(s)/Y(s)$, $i = 0, \ldots, n$. Clearly, any one of the three algorithms can be applied. We now characterize the performance of Generate3. (It is then a simple exercise for the reader to do the same for Generate1 and Generate2.) Note that in this application we have $\bar{a}_i = \mu_i$, $i = 1, \ldots, n$, and $\bar{a}_0 = \lambda$.

THEOREM 4.1 *The long-run acceptance probability for Generate3 for the Jackson network is given by*

$$\lim_{s\to\infty} P(\text{"acceptance at time } s\text{"}) = \frac{\lambda + \sum_{i=1}^{n} \lambda_i + (\sum_{i=1}^{n} \lambda_i(\mu_i - \lambda_i)/\lambda + \sum_{i=1}^{n} \lambda_i)}{\lambda + \sum_{i=1}^{n} \mu_i}.$$

The long-run expected effort for Generate3 for the Jackson network is given by

$$\lim_{s\to\infty} E[V(s)] = \frac{\lambda + \sum_{i=1}^{n} \mu_i}{\lambda + \sum_{i=1}^{n} \lambda_i}.$$

PROOF. Let L_i denote the random variable for the number of customers present at queue i in steady state. It is well known that

$$P(L_1 = l_1, \ldots, L_n = l_n) = \prod_{i=1}^{n} (1 - \rho_i)\rho_i^{l_i}. \tag{7}$$

Since

$$Z = \lambda + \sum_{i=1}^{n} \mu_i 1(L_i > 0),$$

it follows from (7) that

$$E[Z] = \lambda + \sum_{i=1}^{n} \lambda_i$$

and

$$\text{var}[Z] = \sum_{i=1}^{n} \mu_i^2 \rho_i(1 - \rho_i) = \sum_{i=1}^{n} \lambda_i(\mu_i - \lambda_i).$$

Combining these last two expressions for $E[Z]$ and $\text{var}[Z]$ with Theorem 3.3 completes the proof. □

We see from Theorem 4.1 that, if the time average utilization, ρ_i, at each queue i is at least $\frac{1}{2}$ (as is the case in most applications of practical interest), then the long-run acceptance probability is greater than $\frac{1}{2}$. Theorem 4.1 also tells us that it is particularly desirable to have high values of ρ_i for those queues i with high service rates.

Let us now attempt to characterize the performance for some other queuing networks. First, consider the Jackson network described above, but with m_i servers at queue i. Now suppose that $\rho_i = \lambda_i/\mu_i < m_i$. Under these conditions, the associated Markov process is again irreducible, positive recurrent, and aperiodic. In this case $A_0(s) = \lambda$, and $A_i(s)$ takes on values $0, \mu_i, 2\mu_i, \ldots, m_i\mu_i$, $i = 1, \ldots, n$. Following the proof of Theorem 4.1, it can

be shown that the long-run expected effort of Generate3 is given by

$$\lim_{s\to\infty} E[V(s)] = \frac{\lambda + \sum_{i=1}^{n} m_i \mu_i}{\lambda + \sum_{i=1}^{n} \lambda_i}. \tag{8}$$

We leave it to the reader to derive the corresponding expression for the long-run acceptance probability. It is easily shown from (8) that if

$$\frac{\lambda_i}{m_i \mu_i} > \epsilon, \qquad \text{for all} \quad i = 1, \ldots, n, \tag{9}$$

then

$$\lim_{s\to\infty} E[V(s)] \le \frac{1}{\epsilon}$$

for Generate3. Now consider a sequence of queuing networks, where the nth network in the sequence has n stations. If each network in the sequence satisfies (9) for some $\epsilon > 0$, then the long-run average expected effort is $O(1)$.

Second, consider the case of multiple classes, where each class c, $c = 1, \ldots, C$, has an exogenous arrival rate λ^c and has routing probabilities r_{ij}^c, $0 \le i, j \le n$. Suppose that there is one server at each queue, and the service rate is given by μ_i, which does not depend on the class. Also, suppose that the service discipline at each queue is first-come-first-serve (FCFS). There are now C sets of traffic equations; suppose each equation has a nonnegative solution $(\lambda_1^c, \ldots, \lambda_n^c)$. Let $\lambda_i := \lambda_i^1 + \cdots + \lambda_i^C$ and $\lambda := \lambda^1 + \cdots + \lambda^C$, and suppose that $\rho_i := \lambda_i/\mu_i < 1$ for all $i = 1, \ldots, n$. With these modified definitions and assumptions, it is well known that (7) continues to hold true for this multiclass network. Hence, Theorem 4.1 holds unchanged.

Third, consider a multiclass network with a single server at each node, as discussed above, but now suppose that the network is closed; that is, customers neither enter nor leave the network, so that the population size is fixed for each class. The finite-state Markov process associated with this network is irreducible and aperiodic. We now have

$$\lim_{s\to\infty} E[V(s)] = \frac{\sum_{i=1}^{n} \mu_i}{\sum_{i=1}^{n} \mu_i P(L_i > 0)},$$

where L_i is again the number of customers present at node i in steady state. The quantity $P(L_i > 0)$ can be expressed in terms on the defining parameters through "normalization constants."

The networks discussed above are "product-form" queuing networks. Now consider a non-product-form network. In particular, consider the closed, multiclass, single-server network discussed in the paragraph above, but now suppose that the service rate for class c customers at queue i is μ_i^c (i.e., the service rates now depend on the class as well as on the queue). Thus, $A_i(s)$ takes on values $0, \mu_i^1, \ldots, \mu_i^C$, $i = 1, \ldots, n$. The maximum service rate at

queue i is now given by $\bar{\mu}_i$ where $\bar{\mu}_i := \max\{\mu_i^c: c = 1, \ldots, C\}$. Let

$$\gamma_i^c := \lim_{s \to \infty} P(\text{"a class } c \text{ customer is being served at queue } i \text{ in steady state"}).$$

We now have

$$\lim_{s \to \infty} E[V(s)] = \frac{\sum_{i=1}^{n} \bar{\mu}_i}{\sum_{i=1}^{n} \sum_{c=1}^{C} \mu_i^c \gamma_i^c}.$$

Unfortunately, the current tools of queuing theory do not offer a means for expressing γ_i^c in terms of the defining parameters for this non-product-form network. But there are numerous techniques available to approximate these quantities (e.g., see [10]).

5. COMPETING METHODS

We now compare our algorithms with several competing schemes. We carry out the comparison in two contexts: multiserver queuing networks and systems with *similarity*.

5.1 Multiserver Queuing Networks

Consider a network of n queues in series, where each queue is equipped with m servers, each server operates at rate μ, arrivals from the outside arrive at rate λ, and each customer traverses the n queues in sequence. Assume that m is large (say, greater than 100) and that the traffic is moderate to heavy (i.e., $\lambda/m\mu$ is between .5 and .9). We must stress that the complexity bounds given in this section are derived for this specific example; they do not necessarily extrapolate to other examples.

The first competing method discussed makes use of a list of future events. (We assume that the reader is familiar with this standard approach of simulating a discrete-event system.) For this method there is a data structure that contains the time of the next external arrival and the times of the service completions of all the customers currently in the network. Thus, this data structure contains at least 1 and no more than $nm + 1$ events. Assuming that the data structure is organized as a heap, this method requires $O(nm)$ memory and $O(\log n + \log m)$ worst-case effort to generate an event. Under the assumed traffic conditions, the effort will be close to the worst case.

The second competing method also employs a list of future events. But, instead of being implemented across the servers, the future event list is implemented across queues. In this case the data structure contains the time of the next arrival and, for each nonempty queue, the time of the next service completion. Again, assuming that a heap is employed, this method requires $O(n)$ memory and $O(\log n)$ worst-case effort. Note that the complexity bounds for heaps hold for all traffic conditions, not just for the case of moderate to heavy traffic.

The third competing scheme is the algorithm TRANSIT proposed by Fox [7]. Employing Fox's notation, the β_s vector takes the form $\beta_s =$

$(\beta_s(0), \ldots, \beta_s(n))$, where $\beta_s(0) = \lambda$, $\beta_s(j) = \mu l_i$, and l_i is the number of busy servers at queue i at time s. The memory required by this approach is $O(n)$. It can be shown that the long-run expected effort of TRANSIT is $O(n)$. (Indeed, employing the notation of Fox, for any choice of β_0, $|S_i|$ will typically be close to n, and w_i will typically be close to zero; it follows then that the average complexity in this typical situation is $O(n)$.) However, TRANSIT can be modified to have a run time of $O(\log n)$ as is shown in [9], though with a high implicit constant.

References [7] and [9] discuss TRANSIT in the context of being implemented across the queues, as discussed in the paragraph above. However, analogous to the first competing method, TRANSIT can also be implemented across servers (private communication of B. L. Fox, August 1992). The fourth competing scheme is, therefore, TRANSIT implemented over servers, which we refer to as TRANSIT (tailored). In this case the β_s vector takes the form $\beta_s = (\beta_s(0), \ldots, \beta_s(mn))$, where $\beta_s(0) = \lambda$ and where $\beta_s(j)$ equals μ or 0 depending on whether the jth server at time s is busy or not. (We assume that the nm servers are numbered.) Also suppose that β_0 is set to the maximum rates; that is, $\beta_0(0) = \lambda$ and $\beta_0(j) = \mu$, $j = 1, \ldots, mn$. Under these conditions, TRANSIT simplifies:

Algorithm TRANSIT(tailored)

Step 1. Use the alias algorithm to generate a variate J with distribution $q_j = \beta_0(j)/(\beta_0(0) + \cdots + \beta_0(mn))$, $j = 1, \ldots, mn$.

Step 2. If $\beta_s(J) > 0$, accept J, and output J, and quit;
Else go to Step 1.

TRANSIT(tailored) was independently discovered by Fox [8]. Note that TRANSIT(tailored) is just TRANSIT with respect to a majorizing vector. Clearly, TRANSIT(tailored) is an implementation of standard rejection with respect to a nonuniform n-vector. In remarks T10 and T11 of [9], this case of TRANSIT was mentioned; however, its implementation across servers (except for networks of single-server queues) was not noted there but, independently of this paper, was noted in [8]. Note that, in the case $m = 1$, TRANSIT(tailored) and Generate3 are identical. In an analogous fashion, we can define "tailored" algorithms for Generate2 and Generate3. It is easily seen that Generate3(tailored) is equivalent to TRANSIT(tailored), whereas Generate2(tailored) is different.

Mimicking the analysis in Section 4, we see that TRANSIT(tailored) has a long-run expected effort given by

$$\lim_{s \to \infty} E[V(s)] = \frac{\lambda + m\mu n}{\lambda + \lambda n} \leq \frac{m\mu}{\lambda}. \tag{10}$$

Here, $V(s)$ should be interpreted as the number of times Step 1 is invoked in TRANSIT(tailored) when generating an event at time s. It follows from (10) that the long-run expected effort of TRANSIT(tailored) is $O(1)$ for this sequence of models. However, the memory requirements are $O(mn)$.

We therefore arrive at Table II. It is clear that, for these traffic conditions and for a large n, the future event list and TRANSIT are not competitive.

Table II.

	Memory	Long-Run expected effort
Future event list	$O(n)$	$O(\log n)$
TRANSIT	$O(n)$	$O(\log n)$
TRANSIT (tailored)	$O(mn)$	$O(1)$
Generate2, Generate3	$O(n)$	$O(1)$

We must stress that in the complexities of Table II there is an implicit constant factor that depends on the reciprocal of the acceptance probability. The constants for the algorithms in Table II are comparable except for the effort of TRANSIT, whose constant is substantially larger. Compare these constants for TRANSIT(tailored) and Generate3. First, observe that the acceptance probabilities for the two algorithms are identical. However, TRANSIT(tailored) requires one less operation than Generate3 for each pass through the steps of the algorithm. Therefore, the run times of the two algorithms are going to be very close, with TRANSIT(tailored) being slightly faster. The acceptance probability of Generate3 and of TRANSIT(tailored) is greater than that of Generate2, but never more than twice as much if $d = (\sum_{i=0}^{n} \bar{a}_i)/n$.

Based on these observations and the discussion at the end of Section 2, we can make the following conclusions: First, suppose that memory is not an issue. Then TRANSIT(tailored) gives slightly better performance than Generate3. As compared with Generate2, TRANSIT(tailored) requires a few more operations per pass through the steps, but has a higher acceptance probability. However, the acceptance probability of Generate2 can be made arbitrarily close to the acceptance probability of TRANSIT(tailored) by decreasing d. Hence, for sufficiently small d, Generate2 should be slightly faster than TRANSIT(tailored).

Now, suppose that memory is an issue. For Generate2 we suppose that $d = (\sum_{i=0}^{n} \bar{a}_i)/n$ so that the memory requirements of Generate2 and Generate3 are roughly the same. Because TRANSIT(tailored) requires approximately m times the memory required by Generate2 and Generate3, and $m \geq 100$, it is eliminated from the competition. As mentioned at the end of Section 2, there is a trade-off between operations per pass and acceptance probability when comparing Generate2 and Generate3 (although the acceptance probability of Generate2 is greater than one-half that of Generate3). We recommend that the user perform pilot runs when choosing between the two algorithms.

In this example we have assumed that the servers and customers are homogeneous. If this is not the case, for example, if the various servers were in various phases of a PH-distribution, a way to track of the individual servers is needed. For the inhomogeneous case, either a heap implemented across servers (i.e., the first competing scheme) or one of the tailored algorithms would have to be implemented.

5.2 Systems with Similarity

For the definition of similarity, please refer to [7] and [9]. Under similarity, the general version of TRANSIT has $O(1)$ complexity, even when implemented across queues in queuing network settings and/or when similarity is with respect to an n-vector that does not majorize a dynamic rate vector. However, under similarity TRANSIT(tailored), Generate1, Generate2, and Generate3 perform badly for queuing networks where some nodes have light traffic and for the (common) reliability model mentioned in [7]; for the later, arguably, the acceptance probabilities for Generate1, Generate2, and Generate3 are not bounded away from zero as n increases, unless the growth in repair rates and the decrease in failure rates are unnaturally restricted. The point of the general version of TRANSIT is that similiarity does not imply that the reference vector globally majorizes.

6. IMPROVING PERFORMANCE

Although the algorithms Generate2 and Generate3 should be useful for many applications, there are situations for which their performance is less than satisfactory. In particular, unsatisfactory performance occurs when the expected rates are typically far from their respective upper bounds, that is, when $\sum_{i=1}^{n} E[A_i(s)] \ll \sum_{i=1}^{n} \bar{a}_i$. In this section we discuss three modifications of Generate2 that are designed to alleviate or even overcome this problem. In [13] we discussed how some of these modifications can be employed in the efficient simulation of large-scale telephone networks. Throughout this section we simplify the notation by suppressing all references to the time variable s.

Before discussing these modifications, it is beneficial to introduce yet another algorithm for generating variates with changing distributions. The same algorithm has also been independently discovered by Fishman and Yarberry [6], and an equivalent version can be found in [4]. A similar algorithm has also been given in [12, ex. 4.27 and p. 422] in a different context. An informal description of the algorithm follows (for the sake of convenience, suppose that $\log_2 n$ is an integer in this discussion): The algorithm is based on a binary tree with $1 + \log_2 n$ levels and n leaf nodes at the bottom level. The value associated with the ith leaf node is A_i. The value associated with any other node is the sum of the values associated with the two sons of that node. Thus, the value of the node at the root of the binary tree is $Y := A_1 + \cdots + A_n$. Note that $O(n)$ preprocessing time is needed to set up the binary tree. Now, to generate a variate for X, first generate a uniform number, call it Z, over $(0, Y)$. Then compare this number with the value associated with the left child of the root. If it is less, we know that the variate is in $\{1, 2, \ldots, n/2\}$, so proceed with the algorithm on the left side of the binary tree. If it is more, proceed on the right side of the tree with a value Z minus the value stored in the left child. Note that the number of comparisons needed to generate a variate for X is $\log_2 n$. If an A_i changes to an A_i' after generating a variate for X, we can update the binary tree with $O(\log n)$

operations as follows: First, reset the value associated with leaf node i to A_i'. Then, move up the path between this node and the root, and reset the sums accordingly.

In summary, the above "binary-tree algorithm" requires $O(n)$ preprocessing time and $O(\log n)$ time per variate to generate variates for X with changing distributions. The advantage of this algorithm, as compared to Generate2, is that its performance does not depend on how close the A_i's are to their upper bounds. Its disadvantage is that it can take significantly more time to generate a variate for X when n is large and when the acceptance probability for Generate2 is not small.

6.1 Method I: Partitioning

For any subset S of $\{1, \ldots, n\}$, let

$$H(S) := \frac{\sum_{i \in S}(E[A_i]/d)}{\sum_{i \in S}\lfloor \bar{a}_i/d \rfloor},$$

$$\hat{a}(S) := \sum_{i \in S} E[A_i],$$

and

$$Y(S) := \sum_{i \in S} A_i.$$

Now, suppose there exists a partition (S_1, S_2) of $\{1, \ldots, n\}$ with the following properties: (1) $H(S_1)$ is not "small," (2) $H(S_2)$ is "small," and (3) $\hat{a}(S_1)$ is in the "vicinity of " or larger than $\hat{a}(S_2)$. Under these conditions, the following scheme makes sense: First, draw a uniform random variate U. If $U < Y(S_1)/[Y(S_1) + Y(S_2)]$, then we declare that the variate for X belongs to S_1 and use Generate2 (across S_1) to determine it. If $U \geq Y(S_1)/[Y(S_1) + Y(S_2)]$, then we declare that the variate for X belongs to S_2 and use the binary-tree method (across S_2) to find it. Of course, in order to utilize this method, we need to get a handle on $E[A_i]$, $i = 1, \ldots, n$. This can perhaps be done with analytical analysis or with pilot runs.

In the context of queuing networks, the above method may be suitable when a fraction of the queues are in heavy traffic and the remaining queues are in light traffic. We mention here that another partitioning scheme was noted in [8].

6.2 Method II: Pseudo Upper Bounds

Here, for those indices i such that $E[A_i] \ll \bar{a}_i$, we replace $\bar{a}_i$ with a smaller value, perhaps with $E[A_i] + 3\sigma(A_i)$, where $\sigma(A_i)$ is the long-run standard deviation of $A_i(s)$. (This, of course, assumes that one can get a handle on $\sigma(A_i)$, as well as on $E[A_i]$). Now we may have $A_i > \bar{a}_i$ for some indices i. When this occurs, the algorithm Generate2 is no longer correct, since we may

have $B[i] > 1$ for some i. In order to rectify the algorithm, we keep track of the set $\Phi := \{i: A_i > \bar{a}_i\}$ and of the counter $\delta := \max\{A_i/\bar{a}_i: i \in \Phi\}$. (If Φ is empty, set $\delta = 1$.) Then, in Step 3 of Generate2 we replace the test "$R \leq B[I]$" with "$R \leq B[I]/\delta$". We leave it to the reader to verify the correctness of this procedure.

But when is it necessary to update Φ and δ, and how much effort is required for each update? Suppose that an A_i changes to an A_i'. It is only necessary to perform an update in the following circumstances: (1) $A_i/\bar{a}_i \leq 1$ and $A_i'/\bar{a}_i > 1$; (2) $A_i/\bar{a}_i > 1$ and $A_i'/\bar{a}_i > \delta$; (3) $A_i/\bar{a}_i = \delta$ and $A_i'/\bar{a}_i < \delta$; and (4) $A_i/\bar{a}_i > 1$ and $A_i'/\bar{a}_i \leq 1$. In order to minimize the effort to update Φ and δ, Φ can be implemented as a priority queue. With such a data structure, for any one of the four events, $O(\log |\Phi|)$ operations are sufficient for the update. The pseudo upper bounds should be chosen large enough so that $|\Phi|$ and δ are typically small. However, they should not be chosen so small that the acceptance probability becomes undesirably low.

6.3 Method III: Global Updates

For Method II it may not be possible to determine, a priori, good pseudo upper bounds $\bar{a}_i$, $i = 1, \ldots, n$. Or it may be the case that the appropriate choice of pseudo upper bounds changes with the evolution of the underlying process that we are simulating. In these cases we may want to consider occasional global updates of our pseudo upper bounds and reinitializations of the arrays C and B. The reinitialization ensures that all of the entries in B are close to 1. Of course, the process of global update is very costly (requiring $O(n)$ time). But, if this has to be done very rarely, it may be advantageous.

To illustrate, suppose $A_1 + \cdots + A_n$ is small when compared to $\bar{a}_1 + \cdots + \bar{a}_n$. It may be worthwhile to perform a global update. But how do we detect at any given time if Y is low or not? When Y is small, many entries in the array B are much less than 1. We can make use of the following sampling process: Pick, say, $10 \log n$ random outcomes. If a major fraction of these outcomes have an entry much less than, say, $\frac{1}{2}$ in the B array, perform a global update. Using Chernoff bounds we can show that if a major fraction in the sample has a low $B[\ ]$ value, then a major fraction of all the $B[\ ]$ values will be low with high probability. Also, we perform this checking only every $20 \log n$ samples (thus making sure that the total additional cost per sample due to this checking is no more than a fraction). A similar reinitialization was mentioned in [7].

7. CONCLUSIONS

In this paper we have presented three simple algorithms for generating a nonuniform discrete random variate with changing distributions. Under fairly unrestrictive assumptions, these algorithms have an expected $O(1)$ run time. Simulation results [13] confirm the competitiveness of our algorithms in relation to future event schedule and other algorithms now widely in use.

Though our algorithms are useful for many applications, an important open problem is: Does there exist a constant-time algorithm for discrete random

variate generation that does not make any assumptions on the way the rate of $X(s)$ change? Our algorithms are a positive step in this direction.

ACKNOWLEDGMENTS

We are grateful to George Fishman, Bennet Fox, Kurt Mehlhorn, and Marty Reimann for their valuable comments.

REFERENCES

1. ANGLUIN, D., AND VALIANT, L. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst. Sci. 18*, 2 (Apr. 1979), 155–193.
2. BRATLEY, P., FOX, B. L., AND SCHRAGE, L. E. *A Guide to Simulation*. 2nd ed. Springer-Verlag, New York, 1987.
3. CHERNOFF, H. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Stat. 23* (1952), 493–507.
4. DEVROYE, L. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986, 89–90.
5. FISHMAN, G. S. Exploiting special structure to improve future event set management in simulation. Tech. Rep. UNC/OR/TR-90/8, Univ. of North Carolina, Chapel Hill, May 1990.
6. FISHMAN, G. S., AND YARBERRY, L. S. Generating a sample from a k-cell table with changing probabilities in $O(\log_2 k)$ time. Tech. Rep. UNC/OR/TR-90/10, May 1990.
7. FOX, B. L. Generating Markov-chain transitions quickly: I. *Oper. Res. Soc. Am. J. Comput. 2*, 2 (Spring 1990), 126–135.
8. FOX, B. L. Shortening future event lists. *ORSA J. Comput.*, To be published.
9. FOX, B. L., AND YOUNG, A. R. Generating Markov-chain transitions quickly: II. *Oper. Res. Soc. Am. J. Comput. 2*, 1 (Winter 1991), 3–11.
10. HEIDELBERGER, P., AND LAVENBERG, S. S. Computer performance evaluation methodology. *IEEE Trans. Comput. 33*, 12 (Dec. 1984), 1195–1220.
11. KLEINROCK, L. *Queueing Systems, Volume 1: Theory*. Wiley, New York, 1975.
12. MANBER, U. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, Mass., 1989.
13. PRINDIVILLE, M., RAJASEKARAN, S., AND ROSS, K. W. Efficient simulation of large-scale loss networks. Tech. Rep. MS-CIS-91-62, Dept. of Computer Science, Univ. of Pennsylvania, Philadelphia, Aug. 1991.
14. RAJASEKARAN, S., AND REIF, J. H. Derivation of randomized sorting and selection algorithms. Tech. Rep. TR-16-84, Aiken Computing Lab., Harvard Univ., Cambridge, Mass., Mar. 1984.
15. WALRAND, J. *An Introduction to Queueing Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1988.

Received January 1992; revised September and November 1992; accepted December 1992