Efficient Simulation of Large-Scale Loss Networks

Mark Prindiville
University of Pennsylvania

Sanguthevar Rajasekaran
University of Pennsylvania

Keith W. Ross
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/330
For more information, please contact repository@pobox.upenn.edu.
Efficient Simulation of Large-Scale Loss Networks

Abstract
Recently Rajasekaran and Ross [1] presented an algorithm that takes an expected $O(1)$ time to generate a nonuniform discrete random variate. In this paper we discuss how this algorithm can be employed in the efficient simulation of large-scale telephone networks. In a simulation based upon a standard event-list approach, the generation of a new event in the systems take $O(\log n)$ time. With this new algorithm, event generation becomes an $O(1)$ process, and simulation times for large networks can be reduced.

Comments
Efficient Simulation of Large-Scale Loss Networks

MS-CIS-91-62
GRASP LAB 274

Mark Prindiville
Sanguthevar Rajasekaran
Keith W. Ross

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

September 1991
Efficient Simulation of Large-Scale Loss Networks

Mark Prindiville
Dept. of Systems
Univ. of Pennsylvania, Philadelphia, PA 19104

Sanguthavar Rajasekaran
Dept. of Computer and Information Science
Univ. of Pennsylvania, Philadelphia, PA 19104

Keith W. Ross
Dept. of Systems
Univ. of Pennsylvania, Philadelphia, PA 19104

August 21, 1991

ABSTRACT

Recently Rajasekaran and Ross [1] presented an algorithm that takes an expected \(O(1)\) time to generate a nonuniform discrete random variate. In this paper we discuss how this algorithm can be employed in the efficient simulation of large-scale telephone networks. In a simulation based upon a standard event-list approach, the generation of a new event in the system takes \(O(\log n)\) time. With this new algorithm, event generation becomes an \(O(1)\) process, and simulation times for large networks can be reduced.

Key Words: Simulation, Loss Networks, Randomized Algorithms.

---

1 This work has been partially supported by NSF grant NCR-891447.
1 Loss Network Terminology

A network can be defined as a set of nodes, interconnected by links. For a circuit-switched network, the nodes represent physical switches, and each link can be pictured as a "bundle" of circuits. Each circuit is able to handle a single call, making the capacity of a given link the number of circuits which comprise it. If the network is fully-connected, each node is directly-linked to all other nodes.

A given call enters the network at an origination node, and a physical path of connected circuits must be found to reach the destination node. If such a path cannot be found, the call is blocked, or lost. The percentage of calls lost in this manner over a period of time is the network's loss rate over that time.

The paths of calls which are directly-routed traverse only the single link which connects the origination and destination nodes. The paths of calls which are alternately-routed traverse two links, with an intermediate node in the middle. In theory, calls could be routed over three or more links, but carriers are generally unwilling to allocate more than two circuits to a single call. Therefore, they impose a two-hop constraint on network routing.

In fact, when traffic on the network is heavy, and free circuits are scarce, carriers are hesitant to devote even two circuits to a call which cannot be directly-routed. If a carrier were instead to block such a call, it might instead be able to set up two separate directly-routed calls over the same two links, thus doubling its revenue. Therefore, carriers set trunk reservation levels for their links. If the number of free circuits on a link falls to its trunk reservation level, the remaining circuits are reserved for directly-routed calls.

For each incoming call, a routing methodology is necessary to choose a route from the large set of possible routes (which includes the direct route and many alternate routes).

2 Network Definitions

Let us assume that the network to be simulated is fully-connected, with \( N \) nodes. The number \( NP \) of node pairs in the network, then, is:

\[
NP = \binom{N}{2} = \frac{(N^2 - N)}{2}
\]

All node pairs are connected by a link, so the number of links \( J \) is also \( \frac{(N^2 - N)}{2} \). Let each of these links have capacity \( c_j \) and trunk reservation \( \delta_j \).

Define a class-\( j \) call to be a call meant for the node pair directly connected by link \( j \). Let class-\( j \) calls arrive at rate \( \alpha_j \), with mean holding time \( \mu_j \).

Let \( A_j \) be the set of all routes available to class-\( j \) calls. Then \( A_j \) is composed of \( (N - 1) \) elements: the direct route \( \{j\} \) and \( (N - 2) \) alternate routes.

Let \( R \) be the set of all routes in the network. Then \( R \) is composed of \( K \) elements, where:

\[
K = NP \cdot (N - 1) = \frac{(N^3 - 2N^2 + N)}{2}
\]

Let \( R_j \) be the set of routes that pass through link \( j \). Then \( R_j \) is composed of \( T \) elements, where:

\[
T = 1 + 2(N - 2) = 2N - 3.
\]

Let \( n_r \) denote the number of route \( r \) calls in progress, and let \( n = (n_r, r \in R) \) be the state of the network. Then the state space is:

\[
\Omega = \{ n : \sum_{r \in R} n_r \leq c_j, \quad j = 1 \ldots J \} \]
The rate at which events occur in the network is:

\[ \Phi(n) = \sum_{j=1}^{J} \Phi_j(n) \]

\[ \Phi_j(n) = \alpha_j + \sum_{x \in A_j} n_x \mu_j \]

Let \( m_j \) be the number of free circuits on link \( j \). Then:

\[ m_j = c_j - \sum_{x \in R_j} n_x \]

Finally, let \( \{X(s), s=0,1,2,\ldots\} \) be the jump process associated with the Markov process for this network. Also let \( \pi_j = 1 \) if, at the \( s^{th} \) epoch, \( m_j = 0 \) and it is impossible to alternately route a class-\( j \) call under the prevailing methodology. Then an estimate for the probability that a class-\( j \) call is blocked is:

\[ L_j = \frac{\sum_{s=0}^{t} (\pi_j(s) \cdot 1/\Phi(s))}{\sum_{s=0}^{t} (1/\Phi(s))} \]

3 Overviews of DAR-1 and LLR

This paper addresses the efficient simulation of circuit-switched networks under two routing methodologies: Dynamic Alternate Routing (DAR-1) and Least Loaded Routing (LLR).

3.1 The DAR-1 Algorithm

Under DAR-1 routing, a dynamic "routing-list" is maintained. This list contains, for each class of call, an alternate route which serves that class. Let \( AR_j \) be the alternate route currently serving class-\( j \) calls. The algorithm operates in the following manner. When a class-\( j \) call arrives:

1) If the link \( j \) comprising the direct route \( \{j\} \) has a free circuit (i.e. if \( m_j > 0 \)), then the call is set up on that route.

2) If \( m_j = 0 \), then alternate route \( AR_j \), comprised of links \( j_1 \) and \( j_2 \), is tried. If both component links have free circuits in excess of trunk reservation (i.e. if \( m_{j1} > \delta_{j1} \) and \( m_{j2} > \delta_{j2} \)), then the call is set up on \( AR_j \).

3) If the alternate route fails, then the call is blocked. In addition, \( AR_j \) is replaced by choosing at random one of the remaining \((N - 3)\) alternate routes.
3.2 The LLR Algorithm

Let us define the load $l_r$ of a route $r$ as the least number of free circuits available to an alternately-routed call on either of its component links. That is, if route $r$ is composed of links $j_1$ and $j_2$:

$$l_r = \min(m_{j_1} - \delta_{j_1}, m_{j_2} - \delta_{j_2})$$

When a class-$j$ call arrives, the LLR algorithm operates in the following manner:

1) If the link $j$ comprising the direct route $\{j\}$ has a free circuit (i.e. if $m_j > 0$), then the call is set up on that route.

2) If $m_j = 0$, then the call is set up on the alternate route $r'$ that maximizes $l_r$ for $r \in A_j$.

3) If $l_r \leq 0$, there is no alternate route available, and the call is blocked.

4 Overview of the Simulation Process

Let us now examine several methods of efficiently generating a realization of $\{X(s), s=0,1,2,\ldots\}$. Given that $X(s) = n$, how do we obtain subsequent events?

The process of simulating a single event (call arrival or departure) can be partitioned into two smaller, independent processes. In the first process, event generation, we determine the direction (arrival or departure), class, and route of the next event. In the second process, bookkeeping, we update important data structures to account for the new event.

This paper will delineate and compare three methods of event generation: a traditional event-list approach and two approaches which are based on Markov processes and which utilize the generate2 algorithm presented by Rajasekaran and Ross [1]. In the following three sections, each devoted to one of these methods, we will give brief overviews of the methods, estimate the memory required to use them, and derive work estimates for them for both preprocessing time and event processing time.

A final section will discuss efficient handling of the bookkeeping process.

In deriving work estimates for all procedures, we will count both operations and memory accesses. In counting operations, we will not distinguish between comparisons, additions, multiplications. In counting memory accesses, we will not distinguish reads from writes.

4.1 Data Structures Common to All Processes

The following data structures are necessary to all simulation procedures that we will discuss:

1) An array $n$, of dimension $K$, of 2-byte integers to hold the state array $n$. Memory required: $2K$ bytes.

2) A two-dimensional array $LINKS$, of dimension $K$ by 2, of 2-byte integers, to hold the links comprising each route. Memory required: $4K$ bytes.

3) A two-dimensional array $\Gamma$, of dimension $J$ by $T$, of 4-byte integers to hold, for each link $j$, the set $R_j$ of routes that pass through it. Memory required: $(8NJ - 12J)$ bytes.

4) Three arrays: $m$, $\delta$, and $\pi$, of dimension $J$, of 2-byte integers to hold $m_j$, $\delta_j$, and $\pi_j$ for each $j$. Memory required: $6J$ bytes.
5) Two arrays: $\alpha$ and $\mu$, of dimension $J$, of 4-byte reals to hold $\alpha_j$ and $\mu_j$ for each $j$. Memory required: $8J$ bytes.

6) An array $B$ of dimension $J$, of 4-byte reals to hold, for each class $j$, a running total of the numerator of $L_j(t)$, that is:

$$\sum_{s=0}^{t} (\pi_j(s) \cdot 1/\Phi(s))$$

Memory required: $4J$ bytes.

7) Two 4-byte reals $\Phi_n$ and $\Phi$, where $\Phi_n$ holds $\Phi(n)$, and $\Phi$ holds the denominator of $L_j(t)$, that is:

$$\sum_{s=0}^{t} (1/\Phi(s))$$

Memory required: 8 bytes.

The total memory required for these common data structures is $(6K + 8NJ + 6J + 8)$ bytes. For large networks, the first two terms - being $O(N^3)$ - dominate, and the amount of memory required is approximately $7N^3$ bytes.

5 Event-List Approach

5.1 Overview

When a simulation is based on a traditional event-list approach, it revolves around a large data structure called an event-table. This table lists, for each possible type of event in the system, the (simulation) time of its next occurrence. At the start of the simulation, these occurrence times are generated randomly according to relevant probability distributions. The simulation then processes events in order by occurrence time. As each type of event occurs, its occurrence time is reset to the sum of the current (simulation) time and a randomly-generated time.

Applying this approach to our simulation, we note that two types of entries are necessary on the event-table. First, arrival events are necessary for each class of call. Random interarrival times for a class-$j$ arrival may be generated exponentially with parameter $\alpha_j$. Second, at any given time in the simulation, departure entries are necessary for each route serving at least one call. If route $r$ serves class $j$, the parameter $\mu_j$ will be of use in obtaining occurrence times for route $r$ departures.

The maximum number of entries in the event-table is $J + K$.

5.2 Implementation

As documented in the literature, an event-table may be efficiently manipulated if it is stored as a heap, with nodes ordered by occurrence times. We have based our implementation upon the priority queue data structure given in Sedgewick [2]. This implementation stores the heap in an array, with the two children of any node $a$ in array positions $2a$ and $2a+1$.

In the course of the simulation, four processes must be performed on the heap: insertion of an entry, removal of the top node, adjustment of the occurrence time of the top node, and adjustment of the occurrence time of any node.
In order to perform this last operation efficiently, we must be able to quickly find the position
of any node in the heap. Therefore, in addition to maintaining the heap itself, we also maintain an
array with elements containing, for each arrival class and departure route, the positions of their
Corresponding nodes in the heap. Maintenance of this array makes implementation slightly more
complicated than that presented in Sedgewick: when elements in the heap are swapped, the positional
array must also be adjusted.

The algorithms for the four processes all work by first making a structural modification that
may disturb the heap, and then travelling through the heap, reordering elements as necessary.
Pseudo-code relating to these operations is given in Appendix A, with complete derivations of
Corresponding work bounds and estimates.

5.3 Memory Requirements

The event-list approach requires two data structures devoted strictly to event generation:

1) An array of heap entries heap, of dimension \((K + J)\). As we have implemented it, each
heap entry contains a 1-byte integer to hold the direction of the event, a 4-byte integer to
hold the class or route number of the event, and a 4-byte real to hold the occurrence time of
the event. Memory required: \(9 \cdot (K + J)\) bytes.

2) An array event_positions, of dimension \((K + J)\), of 4-byte integers, to hold the positions of
each possible event within the heap. Memory required: \(4 \cdot (K + J)\) bytes.

Thus, the total memory required for event generation under the event-list approach is \(13 \cdot (K + J)\) bytes. For large networks, the \(K\) term - which is \(O(N^3)\) - dominates, and the amount of
memory necessary is approximately \(6.5N^3\) bytes.

5.4 Preprocessing

Before events can begin in the simulation, \(J\) arrival entries must be inserted into the heap - one
for each class of call.

The occurrence time for an arrival to class \(j\) can be generated exponentially with parameter \(\alpha_j\).
The generation of an exponential variable requires the generation of a uniform random variable over
\([0, 1]\), taking the log of that variable, multiplying it by \((-1)\), and dividing it by the parameter. This is
a total of 4 operations (random number, multiplication, logarithm, and division) and 2 memory
accesses (accessing the parameter and writing the result).

Overall, then, with estimates for insertion time taken from appendix A, preprocessing looks
like:

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Est. Operations</th>
<th>Est. Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i \leftarrow 1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>while ((i &lt; J))</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>\quad new time \leftarrow \text{expon}(\alpha_i) \quad</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>\quad \text{insert(arrival, i, new time)}</td>
<td>2.5 \log i + 3</td>
<td>6.5 \log i + 28</td>
</tr>
<tr>
<td>\quad i \leftarrow i + 1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

An estimate for the number of operations necessary in preprocessing is:

\[
E(\text{operations}) = \sum_{i=1}^{J} 2.5 \log i + 9J
\]
An estimate for the number of memory accesses necessary in preprocessing is:

\[ E(\text{accesses}) = 6.5 \sum_{i=1}^{J} \log i + 34J + 1 \]

5.5 Event Processing

For each event generated in the simulation, the event-processing function has four major tasks: 1) to find the direction of the event; 2) to find the class \( j \) of the event; 3) to find the route \( k \) of the event; 4) to adjust the event generation data structures (here the heap and the positional array) to be consistent with the new event.

The first step under the event-list approach is to examine the node at the top of the heap. Three outcomes may then occur: the event may be an accepted arrival, the event may be a blocked arrival, or the event may be a departure.

Let us assume in the following analysis that we have at our disposal the following two functions. Direct route takes a class number and returns the number of the direct route serving that class. Which class takes a route number and returns the class which that route serves. Both of these functions require three operations and three memory accesses.

5.5.1 Accepted Arrivals

If the event is an accepted arrival, (that is, if \( \text{heap1.direction} = 0 \) and \( \pi_{\text{heap1.number}} = 0 \)), then the class \( j \) of the call is simply \( \text{heap1.number} \). If the direct route has free links, (i.e. if \( m_j > 0 \)), then the route \( k \) is \( \{j\} \). Otherwise, the call must be alternately routed. Let \( X_0 \) be the number of operations required to find an alternate route under the prevailing methodology, and let \( X_m \) be the memory accesses required under the prevailing methodology. For DAR, the route \( k \) is \( AR_j \), \( X_0 = 0 \), and \( X_m = 3 \). If LLR is used, all \( (N - 2) \) alternate routes must be examined to see which has the least load \( l_r \). This involves \( 2 \cdot (N - 2) \) subtractions, \( (N - 2) \) comparisons, and roughly \( 4 \cdot (N - 2) \) memory accesses, making \( X_0 = (3N - 6) \) and \( X_m = (4N - 8) \).

After routing is completed, the top node in the heap must have its occurrence time reset with a change top node operation to the sum of the "current" simulation time and a variable exponentially generated with parameter \( a_j \). Finally, the node in the heap corresponding to a departure from route \( k \) must be adjusted. If no such node exists (meaning that this is the only call currently in existence over route \( k \)), then a departure node must be inserted, its occurrence time exponentially generated with parameter \( \mu_j \). Otherwise, the departure time of this new call can be generated as the sum of the current time and a variable exponentially generated with parameter \( \mu_j \). If this new departure time is less than the occurrence time stored in the departure \( r \) node, then that occurrence time is reset to this new value via a change node operation.

5.5.2 Blocked Arrivals

If the event is a blocked arrival, then the only process necessary is to reset, via a change top node operation, the occurrence time of the top node in the heap to the sum of the current simulation time and a variable exponentially generated with parameter \( a_{\text{heap1.number}} \).

5.5.3 Departures

If the event is a departure, (that is, if \( \text{heap1.direction} = 1 \)), then the route \( r \) of the departing call is \( \text{heap1.number} \), and the class \( j \) is the class served by route \( r \). If this is the last call using route \( r \) (that is, if \( n_r = 1 \)), then this top node must be deleted with a remove top operation. Otherwise, the
occurrence time of the top node must be reset with a change top node operation to the sum of the current time and a variable exponentially generated with parameter \((n_f - 1) \cdot \mu_j\).

### 5.5.4 Pseudocode

At any given time, the heap has roughly \((K + J)\) entries, making the operations upon it \(O(\log(K+J))\). For large networks, \(K \gg J\), so we will omit the \(J\) factor in the analysis to follow. In pseudocode, event processing looks like:

<table>
<thead>
<tr>
<th>process</th>
<th>operations</th>
<th>accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>current time \leftarrow heap1.time</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>direction \leftarrow heap1.direction</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

**ACCEPTED ARRIVAL**

If \(heap1.direction = 0\) and \(heap[1].number = 0\)

| j \leftarrow heap1.number | 0 | 2 |
| if \(m_j > 0\) | 1 | 2 |
| | | |
| else | | |
| | | |
| k \leftarrow alternate_route | | |
| new time \leftarrow current time + \text{exponent}(\alpha_j) | 5 | 6 |
| change top(new time) | \(4.5\log K + 1\) | \(11.5\log K + 13\) |
| if \(n_k = 0\) | 1 | 1 |
| | | |
| new time \leftarrow current time + \text{exponent}(\mu_j) | 5 | 6 |
| insert(departure, k, new time) | \(2.5\log K + 3\) | \(6.5\log K + 28\) |
| else | | |
| | | |
| new_time \leftarrow current time + \text{exponent}(\mu_j) | 5 | 6 |
| if new_time < heapdep[k].time | 1 | 4 |
| | | |
| change node(dep_k, new_time) | \(3.5\log K + 2\) | \(9.0\log K + 18.5\) |

**DEPARTURE**

Else if \(heap1.direction = 1\)

| route = heap1.number | 1 | 1 |
| class = which_class(route) | 3 | 3 |
| if \(n_k = 1\) | 1 | 2 |
| | | |
| remove_top() | \(4.5\log K + 3\) | \(11.5\log K + 19\) |
| else | | |
| | | |
| new time \leftarrow cur_time+\text{exponent}(n_k-1 \cdot \mu_j) | 7 | 8 |
| change top(new time) | \(4.5\log K + 1\) | \(11.5\log K + 13\) |

**BLOCKED ARRIVAL**

Else

| class \leftarrow heap1.number | 2 | |
| new time \leftarrow cur_time+\text{exponent}(\alpha\text{heap}[1].number) | 5 | 6 |
| change top(new time) | \(4.5\log K + 1\) | \(11.5\log K + 13\) |
| direction \leftarrow -1 | | 1 |

### 5.5.5 Work Estimates

Let us assume that 50 percent of events are arrivals, that \(p\) is the weighted average blocking probability on the network, and that \(s\) percent of routed calls are routed on alternate routes. It is
obvious from the detailed analysis above that work required in event processing is $O(\log K)$, or $O(\log N)$; what we attempt to do here is to get a rough idea of the constant involved.

To simplify the process of making work estimates, let us assume the usual case for arrivals: that $n_k$ is not 0, and that the new time generated for this new call is not less than the time present at the node. Let us also assume the usual case for departures: that $n_k$ is not 1. The following chart, then, lists the amount of effort necessary for each possible type of event:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>E(operations)</th>
<th>E(memory accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly-routed arrivals</td>
<td>$4.5\log K + 19$</td>
<td>$11.5\log K + 44$</td>
</tr>
<tr>
<td>alternately-routed arrivals</td>
<td>$4.5\log K + 16 + X_0$</td>
<td>$11.5 \log K + 41 + X_m$</td>
</tr>
<tr>
<td>blocked arrivals</td>
<td>$4.5\log K + 9$</td>
<td>$11.5\log K + 30$</td>
</tr>
<tr>
<td>departures</td>
<td>$4.5\log K + 14$</td>
<td>$11.5\log K + 34$</td>
</tr>
</tbody>
</table>

Overall, a rough estimate of the number of operations required to process an event is:

$$E(\text{operations}) = (0.5)(1-s)(1-p)(4.5 \log K + 19) + (0.5)(s)(1-p)(4.5 \log K + 16 + X_0) + (0.5)(p)(4.5 \log K + 9) + (0.5)(4.5 \log K + 14)$$

$$= 4.5 \log K + (0.5)(1-s)(1-p)(19) + (0.5)(s)(1-p)(16 + X_0) + (0.5)(p)(9) + 7$$

A rough estimate of the number of memory accesses is:

$$E(\text{accesses}) = (0.5)(1-s)(1-p)(11.5 \log K + 44) + (0.5)(s)(1-p)(11.5 \log K + 41 + X_m) + (0.5)(p)(11.5 \log K + 30) + (0.5)(11.5 \log K + 34)$$

$$= 11.5 \log K + (0.5)(1-s)(1-p)(44) + (0.5)(s)(1-p)(41 + X_m) + (0.5)(p)(30) + 17$$

To get a feel for the magnitude of these estimates, let's examine a light traffic case. Assume that $p$ is 0.002 and $s$ is 1 percent. Also assume a large network case, with $N = 64$ and $K = 127,008$. Then for DAR:

$$E(\text{operations}) = 4.5\log K + 16 = 92$$

$$E(\text{accesses}) = 11.5\log K + 39 = 234$$
And for LLR:

\[
E(\text{operations}) = 4 \log K + 0.015N + 16 \\
E(\text{accesses}) = 11.5 \log K + 0.02N + 34
\]

In summary, event processing under the event-list approach is O(log K), with a constant of roughly 4.5 for operations and of 11.5 for memory accesses. Since K ≈ N^3, it is also O(log N), with a constant of roughly 13.5 for operations and of 40 for memory accesses.

6 Generate2 Approach

6.1 Overview

It is also possible to generate events in the simulation by generating nonuniform discrete random variates, using a distribution based on the transition probabilities for events in the system. For an arrival to class \( j \), the transition probability is \( \frac{a_j}{\Phi(n)} \), and for a departure from route \( k \) (which serves class \( j \)), the transition probability is \( \frac{n_k \cdot \mu_j}{\Phi(n)} \).

Rajasekaran and Ross discuss efficient methods for generating such random variates in O(1) time. In this part of the paper, we will discuss the direct application of their generate2 algorithm to this simulation.

6.2 Memory Requirements

Let us assume that, in the Rajasekaran/Ross generate2 algorithm, we set \( d \) equal to \( \alpha \). This approach requires four data structures devoted strictly to event generation:

1) An array \( l \), of dimension \( K + J \), of 2-byte integers to hold bucket amounts for arrival and departure events. Memory Required: \( 2 \cdot (K + J) \) bytes.

2) The 4-byte integer \( L \) to hold the total number of buckets, that is:

\[
L = \sum_{i=1}^{K+J} l_i
\]

Memory Required: 4 bytes.

3) An array \( c \), of dimension \( K + J \), of 4-byte reals, as specified in the generate2 algorithm. Memory required: \( 4 \cdot (K + J) \) bytes.

4) An array \( b \), of dimension bounded above by \( 2 \cdot (K + J) \), of 4-byte integers, as specified in the generate2 algorithm. Memory required: \( 8 \cdot (K + J) \) bytes.

Thus, the total memory required for event generation under the generate2 approach is, for a large network, less than 14K bytes, or \( 7N^3 \) bytes.
6.3 Preprocessing

Before events can begin in the simulation, preprocessing must occur as specified in Rajasekaran and Ross, page 7.

STEP 1
In step 1, the array \( l \) must be initialized. First, the following must be computed:

\[
l_j = \text{upper\_floor}[\alpha_j / d] \quad \text{for} \quad j = 1 \ldots J
\]

This requires \( 2J \) operations and \( 3J \) memory accesses. Then, letting route \( k \) serve class \( j \), the following must be computed:

\[
l_{j+k} = \text{upper\_floor}[c_k' \cdot \mu_j / d] \quad \text{for} \quad k = 1 \ldots K, \ j = \text{which\_class}(k)
\]

Here, \( c_k' \) is the minimum number of circuits available to the call across the component links of route \( k \). If \( k \) is the direct route \( \{j\} \), \( c_k' = m_j \). If \( k \) is an alternate route \( \{j_1, j_2\} \), \( c_k' = \min(c_{j_1} - \delta_{j_1}, c_{j_2} - \delta_{j_2}) \). This process requires an additional \( 9K \) operations (upper floor, multiplication, division, 3 operations in which_class, 2 subtractions, and a comparison for each \( k \)) and \( 12K \) memory accesses. Total preprocessing effort in step 1, then, is \((9K + 2J)\) operations and \((12K + 3J)\) memory accesses.

STEP 2
In step 2, the prefix sums of \( l \) must be calculated and put into a temporary array \( \text{temp} \), of dimension \( L \). That is,

\[
\text{temp}_i = \text{temp}_{i-1} + l_i \quad \text{for} \quad i = 1 \to (K + J), \ \text{temp}_0 = 0
\]

This requires \((K + J)\) additions and subtractions, and \(3(K + J)\) memory accesses.

STEP 3
In step 3, the \( b \) array is initialized: cells 1 through \( \text{temp}_1 \) with 1, cells \((\text{temp}_1 + 1)\) through \( \text{temp}_2 \) with 2, etc. This can be accomplished in the following manner:

\[
\begin{array}{l|l|l}
\text{process} & \text{operations} & \text{accesses} \\
\hline
i \leftarrow 1 & 1 & 1 \\
\text{level} \leftarrow 1 & 1 & 2 \\
\text{while} \ i \leq \text{level} & 1 & 3 \\
\quad \text{while} \ i \leq \text{temp} & 1 & 3 \\
\quad b_i \leftarrow \text{level} & 1 & 2 \\
\quad i \leftarrow i + 1 & 1 & 2 \\
\quad \text{level} \leftarrow \text{level} + 1 & 1 & 2 \\
\end{array}
\]

This requires \( L \) comparisons and additions from the inner loop, and \( 7L \) memory accesses from the inner loop. It also requires \((K + J)\) comparisons and additions from the outer loop, and \(4(K + J) + 2\) accesses from the outer loop.

STEP 4
In step 4, the \( C \) array is initialized. First, for the arrival part of the array:

\[
C_j = \lceil \alpha_j / (d \cdot l_j) \rceil \quad \text{for} \quad j = 1 \to J
\]

This requires \( 2J \) operations and \( 4J \) memory accesses. Then, the remaining \((K - J)\) elements of the array - corresponding to departure events - are set to 0. This requires \((K - J)\) memory accesses.
Thus, an estimate of the total preprocessing effort required under generate2 is: $(13K + 8J + 2L)$ operations and $(20K + 13J + 7L + 2)$ memory accesses. Since $L$ is bounded above by $2(K + J)$, for a large network, roughly $17K$ operations and $34K$ memory accesses may be required.

6.4 Event Processing

The first step in event generation under this approach is to apply the generate2 algorithm, obtaining $i$ as a result. If $i \leq J$, then the event is an arrival to class $i$. If the call can be accepted (i.e. if $\pi_j$ is 0), a route $k$ must be chosen for this call as specified in the event-list text, and $C_{J+k}$ must be modified because $n_k$ has changed.

Otherwise, if $i > J$, then the event is a departure from route $i - J$. The class $j$ can be determined from a which_class operation, and $C_{J+k}$ must be modified.

6.5 Pseudocode

<table>
<thead>
<tr>
<th>process</th>
<th>operations</th>
<th>accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>finished $\leftarrow 0$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GENERATE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while $\text{finished} = 0$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>generate $U$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v \leftarrow L \cdot U$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$t \leftarrow \text{upper_floor}(v)$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>if $(t - v) \leq c_b[t]$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$\text{finished} \leftarrow 1$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>result $\leftarrow b_t$</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ARRIVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if $(\text{result} \leq J)$ and $(\pi_{\text{result}} = 0)$</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$\text{direction} \leftarrow 0$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$j \leftarrow \text{result}$</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>if $m_j &gt; 0$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$k \leftarrow \text{first_route}(j)$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k \leftarrow \text{alternate\ route}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_{J+k} \leftarrow [ ((n_k+1) \cdot \mu_j) / d \cdot l_{J+k} ]$</td>
<td>$X_0$</td>
<td>$X_m$</td>
</tr>
<tr>
<td>DEPARTURE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if $\text{result} &gt; J$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$\text{direction} \leftarrow 1$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$k \leftarrow \text{result} - J$</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$j \leftarrow \text{which_class}(k)$</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$c_{J+k} \leftarrow [ ((n_k-1) \cdot \mu_j) / d \cdot l_{J+k} ]$</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>BLOCKED ARRIVAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j \leftarrow \text{result}$</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$\text{direction} \leftarrow -1$</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
6.6 Work Estimates

Let us assume for a moment that the generate2 algorithm requires an average of $Q$ iterations per event to produce random variates. Then generate2 alone contributes $6Q$ operations and $9Q + 5$ memory accesses to the event generation process. The following chart lists the amount of effort necessary for each possible type of event:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>E(operations)</th>
<th>E(memory accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly-routed arrivals</td>
<td>$6Q + 10$</td>
<td>$9Q + 23$</td>
</tr>
<tr>
<td>alternately-routed arrivals</td>
<td>$6Q + X_0 + 7$</td>
<td>$9Q + X_m + 20$</td>
</tr>
<tr>
<td>blocked arrivals</td>
<td>$6Q + 3$</td>
<td>$9Q + 13$</td>
</tr>
<tr>
<td>departures</td>
<td>$6Q + 10$</td>
<td>$9Q + 23$</td>
</tr>
</tbody>
</table>

Recall that $p$ is the weighted average blocking probability in the network, and $s$ is the percentage of routed calls placed on an alternate route. Overall, a rough estimate of the number of operations required to process an event is:

$$E(\text{operations}) = (0.5)(1-s)(1-p)(6Q + 10) +$$

$$+ (0.5)(s)(1-p)(6Q + X_0 + 7) +$$

$$+ (0.5)(p)(6Q + 3) +$$

$$+ (0.5)(6Q + 10)$$

$$= 6Q +$$

$$+ (0.5)(1-s)(1-p)(10) +$$

$$+ (0.5)(s)(1-p)(X_0 + 7) +$$

$$+ (0.5)(p)(3) +$$

$$+ 5$$

A rough estimate of the number of memory accesses is:

$$E(\text{accesses}) =$$

$$+ (0.5)(1-s)(1-p)(9Q + 23) +$$

$$+ (0.5)(s)(1-p)(9Q + X_m + 20) +$$

$$+ (0.5)(p)(9Q + 13) +$$

$$+ (0.5)(9Q + 23)$$

$$= 9Q +$$

$$+ (0.5)(1-s)(1-p)(23) +$$

$$+ (0.5)(s)(1-p)(X_m + 20) +$$

$$+ (0.5)(p)(13) +$$

$$+ 11.5$$

To get a feel for the magnitude of these estimates, let's again examine the light traffic case, with $p$ equal to 0.002 and $s$ equal to 1 percent. Again assume $N = 64$.

Then for DAR:

$$E(\text{operations}) = 6Q + 10$$

$$E(\text{accesses}) = 9Q + 23$$
And for LLR:

\[ E(\text{operations}) = 6Q + .015N + 10 \]

\[ E(\text{accesses}) = 9Q + .02N + 23 \]

Although s, p, and Q are dependent on network parameters, they are not directly dependent on N. Therefore, for the DAR simulation, event generation time is independent of N. It is instead \( O(Q) \), with a constant of roughly 6 for operations and of 9 for memory accesses. For the LLR simulation, the values for \( X_o \) and \( X_m \) - which are derived from the alternate routing process - are \( O(N) \). LLR event generation performance is degraded by an increase in network size.

6.7 Estimate for Q

In any case, the magnitude of Q is of great importance. Let us define \( f_j \), for \( j = 1 \) to \( J \), as the total number of directly-routed class-\( j \) calls in progress. Also define \( d_j \), for \( j = 1 \) to \( J \), as the fraction of routed class-\( j \) calls which are directly-routed.

The events in this system can be partitioned into three types: arrivals, departures from direct routes, and departures from alternate routes. The transition rates *x*, *y* and *z* for these events take the following three forms:

arrivals: \( x_j = \alpha_j \cdot \mu_j \) \( j = 1 \) to \( J \)

direct depts: \( y_j = f_j \cdot \mu_j \) \( j = 1 \) to \( J \)

alternate depts: \( z_k = n_k \cdot \mu_j \) \( k = 1 \) to \( K - J \),

\( k \) serves class \( j \)

Maximum values (in capital letters) for these rates are:

\( X_j = \alpha_j \) \( j = 1 \) to \( J \)

\( Y_j = f_j \cdot \mu_j \) \( j = 1 \) to \( J \)

\( Z_k = n_k \cdot \mu_j \) \( k = 1 \) to \( K - J \),

\( k \) serves class \( j \)

Expected values for these rates are:

\[ E[x_j] = \alpha_j \] \( j = 1 \) to \( J \)

\[ E[y_j] = E[f_j] \cdot \mu_j \] \( j = 1 \) to \( J \)

\[ = d_j \cdot (\alpha_j / \mu_j) \cdot \mu_j \]

\[ = d_j \cdot \alpha_j \]

\[ E[z_k] = (1 - d_j) \cdot (\alpha_j / \mu_j) \cdot \mu_j \]

\[ = (1 - d_j) \cdot \alpha_j \]

\[ k = 1 \ldots (K - J), \]

\( k \) serves class \( j \),

alternately-routed calls spread evenly
Let us assume that \( d \) divides all maximum rates evenly. Then according to Rajasekaran and Ross, page 9:

\[
\mathbb{E}[Q] = \frac{\sum_j x_j}{\sum_j \mathbb{E}[x_j]} + \frac{\sum_j y_j}{\sum_j \mathbb{E}[y_j]} + \frac{\sum_k z_k}{\sum_k \mathbb{E}[z_k]}
\]

\[
= \frac{\sum_j \alpha_j}{\sum_j \alpha_j} + \frac{\sum_j c_j \mu_j}{\sum_j d_j \alpha_j} + \frac{\sum_k c_k \mu_j}{\sum_k (1 - d_j) \alpha_j}
\]

Note that the contributions to the numerator of \( \mathbb{E}[Q] \) from arrival events exactly offset those to the denominator. And for most traffic levels, the contributions from direct departures also roughly offset each other; the number of directly-routed calls for a class will hover close to the maximum allowable.

However, the contribution to the numerator of \( \mathbb{E}[Q] \) from indirect departures far exceeds its contribution to the denominator. Theoretically, many calls are possible across a given alternate route at a given time, but few are realized in practice.

Example

Thus, it is possible that the contributions of indirect departures will drive \( Q \) very high. Consider a sample network which uses DAR, and has \( N = 64 \) and \( c_j = 120, \delta_j = 10, \alpha_j = 102, \mu_j = 1, \) and \( d_j = .99 \) (or \( 1 - s \)) for all \( j \). Here \( J \) is 2016, and \( K \) is 127,008. Then:

\[
\mathbb{E}[Q] = \frac{(2016)(102)}{(2016)(102)} + \frac{(2016)(120)}{(2016)(101)} + \frac{(127,008)(110)}{(127,008)(.016)} \approx 35
\]

\[
\mathbb{E}[\text{operations}] = 6Q + 10 = 220
\]

\[
\mathbb{E}[\text{accesses}] = 9Q + 23 = 338
\]

On the same network, the event-list approach gave us \( \mathbb{E}(\text{operations}) = 92 \), and \( \mathbb{E}(\text{accesses}) = 234 \). In this example, \textit{generate2} performs much worse than the event-list approach.

However, suppose for a moment that we could somehow remove indirect departures from the pool of events generated under \textit{generate2}. Then:

\[
\mathbb{E}[Q] = \frac{(2016)(102)}{(2016)(102)} + \frac{(2016)(120)}{(2016)(101)} \\
\mathbb{E}[Q] = 1.1
\]

\[
\mathbb{E}[\text{operations}] = 6Q + 10 = 17
\]

\[
\mathbb{E}[\text{accesses}] = 9Q + 23 = 33
\]

These expectations are significantly better than those produced by the event-list approach.
7 Partitioning Approach

7.1 Overview

We note above that the inclusion of indirect departure events - with expected rates far from their respective upper bounds - significantly degrades the performance of the generate2 algorithm. However, we can use one of the modifications to generate2 suggested by Rajasekaran and Ross to overcome this problem.

Rajasekaran and Ross define, on page 14, for any subset $S$ of possible events in a system (where the $a_i$'s are the transition rates for the events of $S$):

$$h(S) = \sum_{i \in S} E[a_i]/d \quad \text{and} \quad \sum_{i \in S} \text{upper_floor}[A_i/d]$$

$$a'(S) = \sum_{i \in S} E[a_i]$$

$$a(S) = \sum_{i \in S} a_i$$

If we define $S_1$ to be the subset containing all arrival and direct departure events, and $S_2$ to be the subset containing all indirect departures, we note that (i) $h(S_1) \gg h(S_2)$ and (ii) $a'(S_1) > a'(S_2)$. Rajasekaran and Ross postulate that, in such a case, it makes sense to partition the variate generation algorithm itself. We first draw a uniform variable $U$. If $U < a(S_1) / [a(S_1) + a(S_2)]$, we use generate2 across $S_1$ to determine the event. Otherwise, we use the binary tree method (Raj/Ross, p.14) across $S_2$ to find it.

7.2 Implementation of Binary Tree

Details concerning the implementation of the binary tree method in the partitioning simulation can be found in appendix B, with derivations of work estimates for the two simulation processes which involve the binary tree. The change_tree process, which changes the value of one of the base nodes of the tree and propagates the change through the tree, requires $(5 \log K + 1)$ operations and $(11 \log K + 7)$ memory accesses. The generate_alt_departure process, which draws a random variate and chooses an indirect departure route according to the binary tree method, requires $(5 \log K + 3)$ operations and $(12 \log K + 8)$ memory accesses.

7.3 Memory Requirements

This partitioned approach to event generation requires six data structures devoted strictly to event generation:

1) An array $l$, of dimension $2J$, of 2-byte integers to hold bucket amounts for arrivals and direct departures. Memory required: $4J$ bytes.

2) The 4-byte integer $L$ to hold the total number of buckets, that is:
Memory Required: 4 bytes.

3) An array \( C \), of dimension \( 2J \), of 4-byte reals, as specified in the \textit{generate2} algorithm. Memory required: \( 8J \) bytes.

4) An array \( b \), of dimension bounded by \( 4J \), of 2-byte integers, as specified in \textit{generate2}. Memory required: \( 8J \) bytes.

5) A 4-byte real, \( G \), to hold \( a(S_1) \). Memory required: 4 bytes.

6) An array \( tree \), of approximate dimension \( 2K \), of 4-byte reals to hold the binary tree. Memory required: \( 8K \) bytes.

Thus, the total memory required for event generation under the partitioning approach is, for a large network, roughly \( 8K \) bytes, or \( 4N^3 \) bytes. Note that this is less memory than is required by either the event-list approach (\( 6.5N^3 \) bytes) or the pure \textit{generate2} approach (\( 7N^3 \) bytes).

7.4 Preprocessing

Before events can begin in the simulation, the following steps must occur. The first four steps initialize the \textit{generate2} structures, and the last step initializes the binary tree.

STEP 1

In step 1, the array \( l \) must be initialized. First, the following must be computed:

\[
L = \sum_{j=1}^{J} l_j
\]

Memory Required: 4 bytes.

This requires \( 2J \) operations and \( 3J \) memory accesses. Then, for the direct-departure half of the \( l \) array:

\[
l_j + j = \text{upper}_\text{floor}(c_j \cdot \mu_j / d) \text{ for } j = 1 \ldots J
\]

This process requires an additional \( 3J \) operations (upper floor, multiplication, division) and \( 6J \) memory accesses. Total preprocessing effort in step 1, then, is \( 5J \) operations and \( 9J \) memory accesses.

STEP 2

In step 2, the prefix sums of \( l \) must be calculated and put into a temporary array \( \text{temp} \), of dimension \( L \). That is,

\[
\text{temp}_i = \text{temp}_{i-1} + l_i \text{ for } i = 1 \text{ to } 2J, \text{ temp}_0 = 0
\]

This requires \( 2J \) additions and subtractions, and \( 6J \) memory accesses.

STEP 3

In step 3, the \( b \) array is initialized: cells 1 through \( \text{temp}_1 \) with 1, cells \( (\text{temp}_1 + 1) \) through \( \text{temp}_2 \) with 2, etc. This can be accomplished in the following manner:
This requires \( L \) comparisons and additions from the inner loop, and \( 7L \) memory accesses from the inner loop. It also requires \( 2J \) comparisons and additions from the outer loop, and \( (4 \cdot 2J) + 2 \) accesses from the outer loop.

STEP 4

In step 4, the \( C \) array is initialized. First, for the arrival part of the array:

\[
C_j = \lceil \alpha j \rceil (d_j) \quad \text{for } j = 1 \text{ to } J
\]

This requires \( 2J \) operations and \( 4J \) memory accesses. Then, the remaining \( J \) elements of the array - corresponding to direct-departure events - are set to 0. This requires \( J \) memory accesses.

STEP 5

Finally, all elements of the binary tree must be initialized to 0. There are roughly \( 2K \) elements, so this step requires approximately \( 2K \) memory accesses.

TOTAL EFFORT

Thus, approximately \( (15J + 2L) \) operations and \( (2K + 28J + 7L + 2) \) memory accesses are necessary in preprocessing. Since \( L \) is bounded above by \( 2K \), approximately \( 4K \) (or \( 2N^3 \)) operations and \( 16K \) (or \( 8N^3 \)) memory accesses are required.

7.5 Event Processing

The first step in event generation under the partitioning approach is to draw a uniform variate over \([0, 1]\), and multiply it by \( \Phi \). If the result is less than \( G \), we use the generate2 algorithm to generate an arrival or direct departure. Otherwise, we use the binary tree method to choose an indirect departure.

Suppose that generate2 is used, and that the result is \( i \). If \( i \leq J \), then the event is an arrival to class \( i \). If the call can be accepted (i.e. if \( \pi_i = 0 \)), then a route \( k \) must be found as discussed earlier. If the route chosen is the direct one, then \( C_{j+i} \) must be updated because an extra \( \text{class-}i \) call directly routed now exists. In addition, \( G \) must be increased by \( \mu_j \). If the route chosen is an indirect one, the binary tree must be altered to reflect an extra call on route \( k \). This can be done with a change_tree operation.

If, on the other hand, \( i > J \), the event is a departure from the direct route \( k \) which serves class \( i - J \). \( C_{j+i} \) must be updated because one less directly-routed \( \text{class-}j \) now exists, and \( \mu_j \) must be subtracted from \( G \).

If the binary tree method is used, the procedure generate_alt_departure determines the class and route of the indirect departure. The binary tree must then be altered with a change_tree operation, because an alternate route has one less call.
### 7.6 Pseudocode

<table>
<thead>
<tr>
<th>process</th>
<th>operations</th>
<th>accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate U</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>if $(U \cdot \Phi) &lt; G$</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**GENERATE2**

- `finished \leftarrow 0`
- `while finished = 0`
  - `generate U_2`
  - `v \leftarrow U_2 \cdot L`
  - `t \leftarrow \text{upper_floor}(v)`
  - `if (t - v) \leq C_b[t]`
    - `finished \leftarrow 1`
    - `result \leftarrow b_t`

**ARRIVAL**

- `if result \leq J and \pi_{\text{result}} = 0`
  - `direction \leftarrow 0`
  - `j \leftarrow \text{result}`
  - `if m_j > 0`
    - `k \leftarrow \text{first_route}(j)`
    - `C_{J+j} \leftarrow [(n_k+1) \cdot \mu_j] + [d \cdot l_{J+j}]`
    - `G \leftarrow G + \mu_j`
  - `else`
    - `k \leftarrow \text{alternate route}`
    - `new value \leftarrow (n_k + 1) \cdot \mu_j`
    - `\text{change_tree}(k, \text{new value})`

**DIRECT DEPARTURE**

- `else if result > J`
  - `direction \leftarrow 1`
  - `j \leftarrow \text{result} - J`
  - `k \leftarrow \text{first_route}(j)`
  - `C_{J+j} \leftarrow [(n_k-1) \cdot \mu_j] + [d \cdot l_{J+j}]`
  - `G \leftarrow G - \mu_j`

**BLOCKED ARRIVAL**

- `else`
  - `direction \leftarrow -1`
  - `j \leftarrow \text{result}`

**INDIRECT DEPARTURE**

- `else`
  - `direction \leftarrow 1`
  - `k \leftarrow \text{generate_alt_departure}`
  - `j \leftarrow \text{which_class}(k)`
  - `new value \leftarrow (n_k - 1) \cdot \mu_j`
  - `\text{change_tree}(k, \text{new value})`

---

**Notes:**
- The table entries represent the number of operations and accesses required for each step in the pseudocode.
- The operations and accesses are calculated based on the specific logic and conditions within the pseudocode.
- The `\pi_{\text{result}}` indicates the result of a particular condition or operation.
- The `\text{generate_alt_departure}` and `\text{which_class}(k)` are placeholders for functions or operations that are not explicitly defined in the provided text.
- The `\text{change_tree}(k, \text{new value})` refers to modifying the tree structure with a new value.

---

**References:**
- The pseudocode is structured to handle various conditions and operations efficiently, ensuring that each step is clearly defined and logically sequenced.
- The table format helps in easily identifying the operations and their associated costs, making it easier to understand the overall process.

---

**Understanding:**
- The logic is complex, involving conditions, iterations, and conditional checks to ensure that the desired actions are taken under specific circumstances.
- The pseudocode is designed to manage the generation of U, the generation of U_2, and the handling of events like arrival and departure, ensuring that the system maintains its correct state.

---

**Conclusion:**
- The pseudocode is a robust representation of the system's behavior, allowing for detailed analysis and implementation in practical scenarios.
- The table serves as a useful summary, highlighting the operational costs associated with each step, aiding in the optimization of system performance.
### 7.7 Work Estimates

Let us assume that the *generate2* algorithm used in the partitioning approach requires an average of $Q'$ iterations per event to produce random variates. The following chart lists the amount of effort necessary for each possible type of event:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>E(operations)</th>
<th>E(memory accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly-routed arrivals</td>
<td>$6Q' + 14$</td>
<td>$9Q' + 28$</td>
</tr>
<tr>
<td>alternately-routed arrivals</td>
<td>$6Q' + X_O + 5\log K + 9$</td>
<td>$9Q' + X_m + 11\log K + 26$</td>
</tr>
<tr>
<td>blocked arrivals</td>
<td>$6Q' + 6$</td>
<td>$9Q' + 14$</td>
</tr>
<tr>
<td>direct departures</td>
<td>$6Q' + 14$</td>
<td>$9Q' + 28$</td>
</tr>
<tr>
<td>indirect departures</td>
<td>$10\log K + 12$</td>
<td>$23 \log K + 23$</td>
</tr>
</tbody>
</table>

Recall that $p$ is the weighted average blocking probability in the network, and $s$ is the percentage of routed calls placed on an alternate route. Overall, a rough estimate of the number of operations required to process an event is:

$$
E(\text{operations}) = 
(0.5)(1-s)(1-p)(6Q' + 14) + 
(0.5)(s)(1-p)(6Q' + X_O + 5\log K + 9) + 
(0.5)(p)(6Q' + 6) + 
(0.5)(1-s)(6Q' + 14) + 
(0.5)(s)(10\log K + 12)
$$

A rough estimate of the number of memory accesses is:

$$
E(\text{accesses}) = 
(0.5)(1-s)(1-p)(9Q' + 28) + 
(0.5)(s)(1-p)(9Q' + X_m + 11\log K + 26) + 
(0.5)(p)(9Q' + 14) + 
(0.5)(1-s)(9Q' + 28) + 
(0.5)(s)(23 \log K + 23)
$$

To get a feel for the magnitude of these estimates, let's again examine the light traffic case, with $p$ equal to 0.002 and $s$ equal to 1 percent. Then for DAR:

$$
E(\text{operations}) = 
6Q' + .075\log K + 14 
= 6Q' + 15
$$

$$
E(\text{accesses}) = 
9Q' + .11\log K + 28 
= 9Q' + 30
$$

And for LLR:

$$
E(\text{operations}) = 
6Q' + .075\log K + 14 + .015N 
= 6Q' + 16
$$

$$
E(\text{accesses}) = 
9Q' + .11\log K + 28 + .02N 
= 9Q' + 31
$$
By introducing the partition, we have introduced the log\(K\) terms in these expressions, and our DAR event generation algorithm is no longer strictly independent of \(N\). However, these log\(K\) terms have only minor effect, and our new \(Q'\) is much smaller than the \(Q\) we observed under pure \textit{generate2}. Now \(E[Q']\) is:

\[
E[Q'] = \frac{\sum \alpha_j + \sum c_j \mu_j}{\sum \alpha_j + \sum d_j \alpha_j}
\]

For a network operating under all but the lightest traffic levels, we would expect \(E[Q']\) to range between 1 and 1.5. In our 128 node example:

\[
E[Q'] = \frac{(2016)(102) + (8128)(120)}{(2016)(102) + (8128)(96.9)} = 1.1
\]

\[
E(\text{operations}) = 6 \cdot (1.1) + 15 = 22
\]

\[
E(\text{accesses}) = 9 \cdot (1.1) + 30 = 40
\]

Recalling that the event-list approach expects 92 operations and 234 memory accesses per event, we see that we have substantially reduced the expected effort with our partitioning algorithm.

8 Summary of Event Generation Results

The following chart compares the three event generation routines in terms of memory requirements, approximate preprocessing operations, and approximate event processing operations, when DAR is used.

<table>
<thead>
<tr>
<th>method</th>
<th>memory</th>
<th>preprocessing operations</th>
<th>event processing ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>event-list</td>
<td>6.5(N^3) bytes</td>
<td>4.5(N^2)</td>
<td>12log(N) + ⋯</td>
</tr>
<tr>
<td>generate2</td>
<td>7.0(N^3) bytes</td>
<td>8.5(N^3)</td>
<td>6(Q) + ⋯</td>
</tr>
<tr>
<td>partition</td>
<td>4.0(N^3) bytes</td>
<td>2.0(N^3)</td>
<td>6(Q') + ⋯</td>
</tr>
</tbody>
</table>

Recall that approximately 7\(N^3\) bytes of memory are necessary for storing data structures not specific to event generation. Approximately 11\(N^3\) bytes of memory are necessary to simulate a network of size \(N\) using the partition approach.

We expect that, in most cases, \(Q\) in the chart above will be very large, and \(Q'\) will be close to 1.

If the simulation will span a large number of events, preprocessing time is of little importance to the event generation decision.

9 Bookkeeping

After each event has been generated, the simulation must perform several accounting tasks in reaction to the event:

1) update the proper \(n_k\) value
2) update \(\Phi_n\)
3) update the relevant $m_j$ values
4) increment total simulation time
5) adjust $AR_j$ if necessary
6) adjust the class status array $\pi$ and increment the blocking array $B$

The first three tasks are $O(1)$, and occur only if the event is an accepted arrival or a departure. Assume that the route is $k$, serving class $j$. First, $n_k$ must be incremented or decremented by 1. Second, $\Phi_n$ must be incremented or decremented by $\mu_j$. Third, one or two $m_j$ values - corresponding to the one or two links on the route - must be incremented or decremented.

The fourth task is also $O(1)$, and must always be performed. The running total for simulation "time", kept in the variable $\Phi$, must be incremented by $(1/\Phi_n)$.

If DAR is the prevailing methodology, and the event is a blocked arrival to class $j$, then the old alternate route $AR_j$ must be scrapped, and a new one randomly chosen. This is an $O(1)$ process: a uniform random variate must be chosen over the interval $[1, (N - 2)]$.

9.1 Incrementing the Blocking Array $B$

9.1.1 The Bottleneck

The final task, however, presents a possible bottleneck in the simulation process. Recall that the $B$ array holds a running total of the numerator of $L_j(t)$ - our estimates of blocking probability for each class. That is,

$$B_j = \sum_{s=0}^{t} \left[ \pi_j(s) \cdot \left(1 / \Phi(s) \right) \right]$$

After each event, for each class $j$ in a blocking state, we must set $\pi_j$ to 1 and increment $B_j$ by $1/\Phi_n$.

One way to accomplish this would be to loop through the $J$ classes, checking each to see if it is in blocking state, incrementing $B$ where appropriate. Under DAR, checking a class $j$ requires only referencing $AR_j$, so the process would require $O(J)$, or $O(N^2)$, operations and memory accesses. Under LLR, however, $O(N)$ alternate routes may have to be checked for each class, making the process $O(JN)$, or $O(N^3)$.

Either way, the amount of effort is unsatisfactory. We have engineered an event generation process close to $O(1)$, and we do not want our efforts to be spoiled by an $O(N^2)$ or $O(N^3)$ process in bookkeeping.

9.1.2 Solution: A Linked List

One way to circumvent this problem is to keep a dynamic linked list of classes in blocking state. If the weighted average blocking probability on the network is $p$, then on average the list will have only $(p \cdot J)$ members. When it comes time to increment the $B$ array, we can simply traverse the linked list, adding $\Phi_n$ to the classes appearing on the list. The process is thus reduced to $O(pJ)$.

Obviously, the lighter the traffic, the more effort this approach saves.

However, some effort will also have to be spent maintaining the list. Adding an entry to the list is an $O(1)$ process, but deleting an item from the list is worst case $O(L)$, where $L$ is the number of list elements. Here, $O(L)$ translates to $O(pJ)$, or $O(N^2)$. Moreover, effort will have to be spent determining when to add and delete entries.
9.1.3 List Maintenance

Consider first an accepted arrival. All links on the route (maximum of 2) lose a circuit. We adjust our linked list at only three times: (i) when an \( m_j \) becomes 0, (ii) when an \( m_j \) becomes equal to \( \delta_j \), and (iii) when an arrival is blocked under DAR.

First, if any \( m_j \) becomes 0, the direct route \( \{j\} \) becomes unavailable to class-\( j \) calls. Under DAR, route \( AR_j \) must be checked for availability. If it is unavailable, class \( j \) has entered blocking state. \( \pi_j \) must be set to 1, and class \( j \) added to the list. Thus, the \( (m_j = 0) \) case under DAR is \( O(1) \).

Under LLR, a series of alternate routes (maximum of \( N-2 \)) must be checked for availability. Typically one will be found quickly, and only a few operations will be necessary. In the worst case, no alternate route will be found, \( \pi_j \) must be set to 1, and the class added to the list. The \( (m_j = 0) \) case under LLR is \( O(N) \).

Second, if any \( m_j \) becomes equal to its trunk reservation level \( \delta_j \), some alternate routes that link \( j \) is part of may become infeasible. And one of those suddenly available alternate routes may have been keeping another class of calls nonblocking. For each of the \( O(N) \) routes that link \( j \) is part of, the class \( j_2 \) that each route serves must be checked until a nonblocking route serving it is found. Under DAR, these checks are \( O(1) \), and the \( (m_j = \delta_j) \) case is \( O(N) \). Under LLR, the checks themselves are possibly \( O(N) \), making this \( O(N^2) \) in the worst case.

Finally, if an arrival is blocked under DAR, \( AR_j \) is changed, as noted above. If the new alternate route chosen randomly can accept an alternately routed call, then class \( j \) must be deleted from the list, a worst case \( O(N^2) \) process.

Although these maintenance procedures are worst case \( O(N^2) \), the worst cases are somewhat rare - especially in light traffic. Often, the route will be composed of only one link, \( m_j \) will be neither 0 nor \( \delta_j \), and list maintenance will require only two comparisons.

Consider next a departure. All links on the route (maximum of 2) gain a circuit. We adjust our linked list at only two times: (i) when an \( m_j \) value becomes 1, and (ii) when an \( m_j \) value becomes \((\delta_j + 1)\).

First, when an \( m_j \) value becomes 1, a direct route has been freed. If \( \pi_j \) is 1, it is changed to 0, and class \( j \) is removed from the blocking list. This is worst case \( O(p) \), or \( O(pN^2) \).

Second, if any \( m_j \) becomes equal to \((\delta_j + 1)\), alternate routes using link \( j \) may have been freed. Each of the \( O(N) \) routes using the link must have its class checked to see if the class is on the blocking list. In light traffic, most will not be. But if any are, they must be checked to see if they are still blocking. Under DAR, these checks are \( O(1) \). Under LLR, they are worst case \( O(N) \). If any class may indeed be deleted from the blocking list, the deletion, as noted, is an \( O(N^2) \) process.

As in the arrival case, list maintenance after a departure is worst case \( O(N^2) \), but will generally require just two comparisons.

10 Computational Results

To test our analysis of the event generation process, we have written a test program which simulates a loss network under DAR or LLR routing. The simulation can generate events by using any of the three methods covered in this paper: an event-list approach, the pure generate2 algorithm, or the partitioned approach.

10.1 Summary of Expectations

We have chosen as a test network the 64-node example used in the preceding analysis. All 2,016 links on this network have capacity \( c = 120 \), and reservation \( \delta = 10 \). This network has 2,016 classes of calls, each with mean holding time \( \mu = 1 \). For DAR trials, we use arrival rate \( \alpha = 102 \), while for LLR trials we use \( \alpha = 107.5 \). We have chosen these levels to make the loss rate \( p \) in the network approximately 0.002 and the fraction \( s \) of routed calls which are alternately routed approximately 0.01.

Our expectations for this sample network, as developed previously, are:
However, two factors limit the precision of these expectations. First, the work estimates for the heap routines in the event-list simulation are fairly rough. In Appendix A, to simplify the analysis, we assume that half of the maximum logK iterations are necessary in all routines. However, this is not strictly the case. In the change_node operation, for example, it is likely that, on average, less than half of the maximum number of iterations will be necessary. Second, we have not been extremely precise in counting operations and memory accesses. Though we have placed additions, multiplications, upper_floor operations, etc. in a single category, they require different amounts of CPU time.

10.2 Summary of Results

We have obtained the following results from our simulation program for this 64-node case. The simulations cover 5,000,000 events each. All times are in seconds.

<table>
<thead>
<tr>
<th>simulation type</th>
<th>routing algorithm</th>
<th>generation algorithm</th>
<th>E(Q)</th>
<th>E(operations)</th>
<th>E(accesses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DAR</td>
<td>event-list</td>
<td>-</td>
<td>92</td>
<td>234</td>
</tr>
<tr>
<td>2</td>
<td>LLR</td>
<td>event-list</td>
<td>-</td>
<td>93</td>
<td>235</td>
</tr>
<tr>
<td>3</td>
<td>DAR</td>
<td>generate2</td>
<td>35</td>
<td>220</td>
<td>338</td>
</tr>
<tr>
<td>4</td>
<td>LLR</td>
<td>generate2</td>
<td>33</td>
<td>209</td>
<td>321</td>
</tr>
<tr>
<td>5</td>
<td>DAR</td>
<td>partition</td>
<td>1.1</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>LLR</td>
<td>partition</td>
<td>1.1</td>
<td>23</td>
<td>41</td>
</tr>
</tbody>
</table>

The second column lists the empirical values for Q obtained in the simulations. These values are quite close to the expected values.

The third column lists empirical values for event generation times. As predicted, the partitioning approach is the best performer, and pure generate2 the worst.

The event-list approach performs slightly better than expected; our work estimates appear to be overstated. Based on these estimates, we expect partitioning to outperform the event-list approach by about a factor of 4; instead, the factor is only about 2. We expect the event-list approach to outperform generate2 by a factor of 2.4; the actual factor is roughly 5.

We also expect partitioning to outperform pure generate2 by a factor of 10; the empirical factor is roughly 11. This close agreement suggests that, although the event-list work expectations are slightly overstated, the expectations for the other approaches are fairly accurate.

The fourth column lists bookkeeping times for all simulations. Five million events are insufficient to put the network in steady state, and blocking is slightly higher in the event-list simulations than in the others. Therefore, bookkeeping times in event-list simulations exceed those in other simulations.
The fifth column lists event-generation preprocessing times for the simulations, along with total preprocessing times. The empirical times for preprocessing for event generation agree with the predictions made on page 21. Partitioning involves the least preprocessing, followed by the event-list approach and then pure generate2. However, in a large-scale simulation such as this one, preprocessing for event generation is insignificant.

The total times listed for preprocessing include the $O(N^3)$ time necessary to set up the $LINKS$ and $r$ arrays used in bookkeeping. The amounts in the table appear significant, but they remain fixed as the number of events to be simulated is increased. If we were simulating to obtain precise results, rather than to analyze the efficiency of the simulation, we would simulate for more than five million events, and preprocessing would appear less significant. Moreover, once created, these arrays can be stored and reused in subsequent runs (based on $N$-node networks); such preprocessing is not necessary for each run.

10.3 Is Partitioning $O(1)$?

We claim in the analysis that the partitioning algorithm requires an amount of work largely independent of network size. We have tested this assertion by simulating a 32 node network with parameters similar to those used in the 64 node case. We have obtained the following results:

<table>
<thead>
<tr>
<th>$N$</th>
<th>Event Generation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>422</td>
</tr>
<tr>
<td>64</td>
<td>420</td>
</tr>
</tbody>
</table>

These results seem to confirm that event generation under partitioning is $O(1)$.

For very large networks ($N \approx 128$), implementing the partitioning approach rather than a traditional $O(\log N)$ event-list approach would greatly reduce event generation time. However, the worst-case $O(N^2)$ bookkeeping process and, to some extent, the preprocessing function would become performance bottlenecks.
References


Appendix A: Work Estimates for Event-List Heap Routines

In this appendix, we assume that the heap in question has \( H \) elements. We assume that \( heap_h \), for \( h = 1 \) to \( H \), is a node in the heap which possesses three fields. \( Heap_h.direction \) is 0 if node \( h \) is an arrival node, and 1 if the node is a departure. \( Heap_h.number \) is the class number of the event if it is an arrival, or the route number if it is a departure. \( Heap_h.time \) is the occurrence time for node \( h \).

Because this data structure is a heap ordered on occurrence time, \( heap_h.time \) must always be less than or equal to the occurrence times of children \( heap_{2h}.time \) and \( heap_{2h+1}.time \).

We assume that \( arr_j \), for \( j = 1 \) to \( J \), holds the position in the heap of the node corresponding to an arrival to class \( j \), and that \( dep_k \), for \( k = 1 \) to \( K \), holds the position in the heap of the node corresponding to a departure from route \( k \).

The algorithms for the four heap processes (insertion, top node removal, top node adjustment, adjustment of any node) all work by first making a structural modification that may disturb the heap, and then travelling through the heap, reordering elements as necessary. The two fundamental travelling processes are \( upheap \) and \( downheap \). \( Upheap(i) \) starts at node \( i \) and traverses to the top of the heap, rearranging nodes to fix inconsistencies. \( Downheap(i) \) starts at node \( i \) and traverses to the bottom of the heap, also patching inconsistencies.

**A.1 Procedure Upheap(i)**

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Operations</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( heap_0.time \leftarrow -1 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( v \leftarrow heap_i )</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{while } heap_i/2.time \geq v.time \\
\quad heap_i \leftarrow heap_i/2 \\
\quad \text{if } heap_i.direction = 0 \\
\quad \quad arr_{heap[i].number} \leftarrow i \\
\quad \text{else} \\
\quad \quad dep_{heap[i].number} \leftarrow i \\
\quad \quad i \leftarrow i + 2 \\
\quad heap_i \leftarrow v \\
\quad \text{if } heap_i/2.direction = 0 \\
\quad \quad arr_{heap[i].number} \leftarrow i \\
\quad \text{else} \\
\quad \quad dep_{heap[i].number} \leftarrow i
\]

Since the inner while loop runs for a maximum of \( \log H \) iterations, upper bounds on the work required for the \( upheap \) process are: \( (5 \log H + 1) \) operations and \( (13 \log H + 12) \) memory accesses. On average, let us postulate that half of the \( \log H \) iterations will be necessary. An estimate, then, of the work required for an upheap operation is \( (2.5 \log H + 1) \) operations and \( (6.5 \log H + 12) \) memory accesses.

**A.2 Procedure Downheap(i)**

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Operations</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v \leftarrow heap_i )</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Since the inner while loop runs for a maximum of \( \log H \) iterations, upper bounds on the work required for the \( downheap \) process are: \( (5 \log H + 1) \) operations and \( (13 \log H + 12) \) memory accesses. On average, let us postulate that half of the \( \log H \) iterations will be necessary. An estimate, then, of the work required for a downheap operation is \( (2.5 \log H + 1) \) operations and \( (6.5 \log H + 12) \) memory accesses.
while \( i \leq \frac{H}{2} \)

\[
\begin{align*}
\text{z} & \leftarrow 2 \cdot i \\
\text{if} \; \text{z} \leq H \text{ and } \text{heap}_z\text{.time} \leq \text{heap}_{z+1}\text{.time} \; & \rightarrow \; 3 \\
\text{z} & \leftarrow z + 1 \\
\text{if} \; \text{v}\text{.time} \leq \text{heap}_z\text{.time} & \leftarrow \; 1 \\
\text{break out of while loop} \\
\text{heap}_1 & \leftarrow \text{heap}_z \\
\text{if} \; \text{heap}_1\text{.direction} = 0 & \rightarrow \; 1 \\
\text{arr}_\text{heap}[i]\text{.number} & \leftarrow i \\
\text{else} & \rightarrow \; 3 \\
\text{de}_\text{heap}[i]\text{.number} & \leftarrow i \\
\text{i} & \leftarrow z
\end{align*}
\]

\[
\begin{align*}
\text{heap}_1 & \leftarrow v \\
\text{if} \; \text{heap}_1\text{.direction} = 0 & \rightarrow \; 1 \\
\text{arr}_\text{heap}[i]\text{.number} & \leftarrow i \\
\text{else} & \rightarrow \; 3 \\
\text{de}_\text{heap}[i]\text{.number} & \leftarrow i
\end{align*}
\]

Since the inner while loop runs for a maximum of \( \log H \) iterations, upper bounds on the work required for the \textit{downheap} process are: \((9 \log H + 1)\) operations and \((23 \log H + 11)\) memory accesses. Again postulating that half of the maximum number of iterations are necessary on average, work estimates for a \textit{downheap} procedure are \((4.5 \log H + 1)\) operations and \((11.5 \log H + 11)\) memory accesses.

\section*{A.3 Procedure Insert}

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Max. Operations</th>
<th>Memory Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H \leftarrow H + 1 )</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>\text{heap}_H\text{.time} \leftarrow \text{new occurrence time}</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>\text{heap}_H\text{.direction} \leftarrow \text{new direction}</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>\text{heap}_H\text{.number} \leftarrow \text{new number}</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
| if \text{heap}_H\text{.direction} = 0 & \rightarrow \; 1 \\
\text{arr}_\text{heap}[H]\text{.number} & \leftarrow H \\
\text{else} & \rightarrow \; 3 \\
\text{de}_\text{heap}[H]\text{.number} & \leftarrow H \\
\text{upheap}(H) & \rightarrow \; 5 \log H + 1 \\
\text{13 logH + 12} |

Thus, an insert operation requires less than \((5 \log H + 3)\) operations and \((13 \log H + 28)\) memory accesses. A work estimate for these numbers: \((2.5 \log H + 3)\) operations and \((6.5 \log H + 28)\) memory accesses.

\section*{A.4 Procedure Remove Top Node}

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>Max Operations</th>
<th>Memory Accesses</th>
</tr>
</thead>
</table>
| if \text{heap}_1\text{.direction} = 0 & \rightarrow \; 1 \\
\text{arr}_\text{heap}[1]\text{.number} & \leftarrow -1 \\
\text{else} & \rightarrow \; 2 |

28
Thus, a remove-top operation requires less than \((9 \log H + 3)\) operations and \((23 \log H + 19)\) memory accesses. A work estimate for these numbers: \((4.5 \log H + 3)\) operations and \((11.5 \log H + 19)\) memory accesses.

A.5 Procedure Change Top Node(new value)

Pseudocode

\[
\begin{align*}
\text{heap}.time &= \text{new value} \\
\text{downheap}(1)
\end{align*}
\]

Thus, a replace-top operation requires less than \((9 \log H + 1)\) operations and \((23 \log H + 13)\) memory accesses. A work estimate for these numbers: \((4.5 \log H + 1)\) operations and \((11.5 \log H + 13)\) memory accesses.

A.6 Procedure Change Node(position, new time)

Pseudocode

\[
\begin{align*}
\text{old time} &= \text{heapposition.time} \\
\text{heapposition.time} &= \text{new time} \\
\text{if new time} &< \text{old time} \\
\quad \text{upheap(position)} \\
\text{else} \\
\quad \text{downheap(position)}
\end{align*}
\]

Thus, a change-node operation requires less than \((9 \log H + 2)\) operations and \((23 \log H + 19)\) memory accesses. Let us assume that the new time will exceed the old time with probability 0.5. Then work estimates for this procedure are \((3.5 \log H + 2)\) operations and \((9 \log H + 18.5)\) memory accesses.

A.7 Summary of Work Estimates for the 4 Processes

\[
\begin{array}{|l|c|c|}
\hline
\text{Process} & \text{Estimated Operations} & \text{Estimated Memory Accesses} \\
\hline
\text{insert} & 2.5 \log H + 3 & 6.5 \log H + 28 \\
\text{remove top} & 4.5 \log H + 3 & 11.5 \log H + 19 \\
\text{change top} & 4.5 \log H + 1 & 11.5 \log H + 13 \\
\text{change node} & 3.5 \log H + 2 & 9.0 \log H + 18.5 \\
\hline
\end{array}
\]
Appendix B: Work Estimates for the Binary Tree Routines

Tree Structure

We have implemented the binary tree as an array with the children of node \( a \) located in positions \( 2a \) and \( 2a + 1 \). The number of levels \( UL \) above the base level in the tree is:

\[
UL = \text{upper}_\text{floor} \lfloor \log(K - 1) \rfloor
\]

The number \( BN \) of node necessary on the base level, then, is:

\[
BN = 2^{UL}
\]

The total number \( T \) of nodes in the tree, then, is:

\[
T = (2 \cdot BN) - 1
\]

The top of the tree is at position 1. The base nodes consist of positions \( (T / 2) \) to \( T \), which contain the call amounts of routes 1 through \( K \). Let \( RO = (T / 2) \) denote the position in the tree where the base nodes begin.

Operations

There are two operations necessary to the partitioning simulation which involve the binary tree. The change_tree operation changes the call amount stored at a base node and propagates this change up the tree. The generate_alt_departure process draws a random variate and, using the process described in Rajasekaran and Ross on page 14, chooses an indirect departure route.

B.1 Procedure Change_Tree(route, new value)

<table>
<thead>
<tr>
<th>process</th>
<th>operations</th>
<th>accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos (-) RO + route</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>level (-) 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>old value (-) treepos</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>( \text{while} ) level ( \leq ) UL</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
| \hline
| \( \text{tree} \)pos \(-\) treepos - oldvalue + newvalue | 2          | 5        |
| level \(-\) level + 1 | 1          | 2        |
| pos \(-\) pos + 2   | 1          | 2        |

Total operations for a change_tree operation: \( 5\log K + 1 \). Total memory accesses: \( 11\log K + 7 \).

B.2 Procedure Generate_Alt_Departure

<table>
<thead>
<tr>
<th>process</th>
<th>operations</th>
<th>accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>i (-) 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>pos (-) 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>tot (-) 0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
generate U
v ← tree1 ⋅ U

while i ≤ UL
   pos ← pos ⋅ 2
   if v > treepos + tot
      tot ← tot + treepos
   pos ← pos + tot

return k ← pos - RO

Total operations for a generate_alt_departure operation: $5\log K + 3$. Total memory accesses: $12\log K + 8$. 