Maintaining Discrete Probability Distributions Optimally*

Torben Hagerup,[1] Kurt Mehlhorn,[1] and J. Ian Munro[2]

[1] Max-Planck-Institut für Informatik, Im Stadtwald, W-6600 Saarbrücken, Germany.
[2] Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

Abstract. Consider a *distribution* as an abstract data type that represents a probability distribution f on a finite set and supports a *generate* operation, which returns a random value distributed according to f and independent of the values returned by previous calls. We study the implementation of *dynamic* distributions, which additionally support changes to the probability distribution through *update* operations, and show how to realize distributions on $\{1, \ldots, n\}$ with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time. We also consider *generalized distributions*, whose values need not sum to 1, and obtain similar results.

1 Introduction

The problem of generating sequences of independent random values with a specified discrete probability distribution is fundamental in the area of discrete-event simulation [BFS87]. If the probability distribution is invariant over time, the well-known *alias method* [BFS87, Section 5.2.8] can generate each successive value in constant time (following $O(n)$ preprocessing time). We generalize this to a dynamic setting, where the probability distribution is allowed to change over time, extending work of Fox [Fox90] and Rajasekaran and Ross [RR91] and complementing work of Matias, Vitter, and Ni [MVN93]. Our results lead to more efficient simulations of an important class of queueing networks, the so-called open Jackson networks (see [Kle75, Section 4.8] and [Fox90]).

A *distribution* f on a finite set D assigns a probability to each element of D, i.e., f is a nonnegative function defined on D with $\sum_{x \in D} f(x) = 1$. We study the problem of maintaining a distribution f on a finite domain D under the operations *generate*, which returns a random value distributed according to f and independent of the values generated by previous calls, and *update*, which shifts a specified amount of probability from one element of D to another. Our first result is

Theorem 1. *Distributions on $\{1, \ldots, n\}$ can be maintained with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time.*

In some applications, notably the simulation of open Jackson networks mentioned above, it is more natural to view a distribution on a set D as induced by an arbitrary nonnegative function f on D, whereby the probability of each $x \in D$ is proportional to $f(x)$. We model this situation by defining a *generalized distribution* on a finite domain D as any nonnegative function f on D. The values of f are called *weights*, and $W = \sum_{x \in D} f(x)$ is the *total weight* of f. A *generate* operation produces each element $x \in D$ with probability proportional to $f(x)$, i.e., with probability $f(x)/W$, and an *update*

* Supported by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II). The research was carried out in part while the first author was with the Departament de LSI of the Universitat Politècnica de Catalunya in Barcelona, Spain, and in part while the third author was visiting the MPI für Informatik.

operation changes a single weight arbitrarily. A generalized distribution on a set of size n is called *polynomially-bounded* if its weights remain integral and bounded by a fixed polynomial in n. We show

Theorem 2. *Polynomially-bounded generalized distributions on $\{1,\ldots,n\}$ can be maintained with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time.*

Our results are valid in the standard randomized real RAM model. More specifically, we assume that the following operations take constant time: arithmetic (addition, subtraction, multiplication, and integer division) on integers of absolute value polynomial in n, real arithmetic and the floor function, and drawing a random number from the uniform distribution on the interval $[0,1]$.

Theorem 1 completely settles the problem of maintaining a distribution. For generalized distributions we achieve optimality only under the assumption that the weights are integral and polynomially bounded; this is definitely the setting of greatest practical relevance. Matias, Vitter, and Ni [MVN93] recently showed in independent work that arbitrary generalized distributions can be maintained with $O(\log^* n)$ expected update and generate times in a more powerful model that assumes constant-time operations on integers of arbitrary size. A combination of their original methods and those described here yields a solution in the more powerful model with constant expected update and generate times [MVN93]. Previous to [MVN93] and the present paper, the only method that could deal with general update operations and put no restrictions on the (generalized) distribution had generate and update times of $\Theta(\log n)$ (store the weights at the leaves of a complete binary tree, with each internal node storing the total weight at its leaf descendants). Sublogarithmic methods were only known under the assumption that there are known and fixed generalized distributions $\underline{f}$ and $\overline{f}$ such that always $\underline{f}(x) \leq f(x) \leq \overline{f}(x)$, for all $x \in D$. Under this assumption, Rajasekaran and Ross [RR91] achieve update and generate times of $O(\overline{\alpha}/\underline{\alpha})$, where $\underline{\alpha} = (1/n)\sum_{x\in D}\underline{f}(x)$ and $\overline{\alpha} = (1/n)\sum_{x\in D}\overline{f}(x)$.

2 Maintaining Distributions

2.1 Two Simple Methods

The problem of generating a random value distributed according to a generalized distribution f on $\{1,\ldots,n\}$ has a trivial $O(n)$-time solution whose correctness is obvious: Compute the prefix sums of f, i.e., let $s_i = \sum_{j=1}^{i} f(j)$, for $i = 0,\ldots,n$, and then pick a random value z from the uniform distribution on $(0, s_n]$ and return the unique $i \in \{1,\ldots,n\}$ with $s_{i-1} < z \leq s_i$.

Lemma 1. *Given a generalized distribution f on $\{1,\ldots,n\}$, a random value distributed according to f can be generated in $O(n)$ time using $O(n)$ space.*

The algorithm described above will be called the *prefix-sums algorithm* or the naive algorithm. We next develop another algorithm, based on the *rejection method* [BFS87, Section 5.2.5], that performs efficiently in a very special case, namely for so-called *flat generalized distributions.* A generalized distribution is *flat* if its nonzero weights all lie in $[r, 2r]$, for some known $r > 0$.

First consider the following problem: Given a real number p with $0 \leq p \leq 1$, accept with probability p and reject with probability $1-p$; here accepting and rejecting are symbolic actions without any concrete meaning. The problem is trivial to solve in constant time: Simply draw a random value z from the uniform distribution on $[0,1]$ and accept if and only if $z \leq p$. When p has been fixed, an algorithm with this behavior will be called a p-acceptor.

Lemma 2. *A generalized distribution f on $\{1,\ldots,n\}$ whose nonzero weights lie in the interval $[r,2r]$, for some known (but possibly time-dependent) $r>0$, can be maintained with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time.*

Proof. Store the elements of $D'=\{j : 1\le j\le n \text{ and } f(j)>0\}$ in the first $|D'|$ cells of an array $A[1\,..\,n]$, and also record $n'=|D'|$. For $j=1,\ldots,n$, store in the jth cell of another array $B[1\,..\,n]$ the weight $f(j)$ and, if $j\in D'$, the position of j in A. In order to generate a random value distributed according to f, first pick a random value l from the uniform distribution on $\{1,\ldots,n'\}$. Then convert l to a random element j of D' by indexing into A, i.e., let $j=A[l]$, and run an $f(j)/(2r)$-acceptor. If the acceptor accepts, output the value j; otherwise restart the algorithm, i.e., pick a new random value l, etc.

It is easy to see that the value j is output with probability proportional to $f(j)$, for $j=1,\ldots,n$, i.e., the algorithm is correct. The algorithm carries out successive trials until some trial succeeds. Each trial can be executed in constant time and succeeds with probability $(1/n')\sum_{j\in D'} f(j)/(2r)\ge 1/2$. The required number of trials is therefore geometrically distributed with expected value at most 2.

An update is straightforward, except when n' changes. When D' gains an element, store it in the first unused cell in A and increment n'; when D' loses an element, exchange it with the element stored in the last used cell in A and decrement n'. □

2.2 Reducing the Domain Size

In this subsection we discuss a simple way of combining data structures designed to maintain different "parts" of a single generalized distribution. It can be viewed as a dynamic version of the *composition method* [BFS87, Section 5.2.7].

It is convenient to divide the universe of weights into discrete *ranges*: For every integer i, let $R_i=(2^i,2^{i+1}]$. Given a generalized distribution f on a domain D and an integer i, we call $D_i=\{j\in D : f(j)\in R_i\}$ the ith *layer domain* of f and the restriction of f to D_i the ith *layer* of f. Since each layer of a generalized distribution is flat, we know from Lemma 2 that any given layer can be maintained with constant expected generate time and constant update time. Our goal now is to put together solutions for different layers in order to obtain an overall solution. The idea is quite simple: To generate a random value, first somehow decide on a layer, and then generate the value using the data structure for that layer. "Somehow deciding on a layer" in fact again is generating a value according to a given generalized distribution, which now assigns weight $\sum_{j\in D_i} f(j)$ to i, for all integers i — we call it the *layer distribution.*

To *locate* an element $x\in D$ with respect to a partition $D_1,\ldots,D_m$ of a set D is to determine the unique $i\in\{1,\ldots,m\}$ with $x\in D_i$. If the partition $D_1,\ldots,D_m$ is successively changed by a sequence of updates, it will be called *slowly dynamic* if each update causes only a constant number of elements of D to move between sets in the partition, and if the set of moving elements can be determined in constant time.

Suppose now that f is a distribution on $D=\{1,\ldots,n\}$, and that $D_1,\ldots,D_m$ is a slowly dynamic partition of D such that any $x\in D$ can be located with respect to $D_1,\ldots,D_m$ in constant time. The *dynamic composition method* applied to f and $D_1,\ldots,D_m$ maintains the restriction f_i of f to D_i, for $i=1,\ldots,m$, as well as the *group distribution* g on $\{1,\ldots,m\}$ with $g(i)=W_i=\sum_{x\in D_i} f(x)$, for $i=1,\ldots,m$. To generate a random value z according to f, first select a group D_i by choosing i according to the group distribution g, and then generate z according to f_i. Since we choose between groups with the correct probabilities and between the elements of the chosen group with the correct relative probabilities, z is indeed distributed according to f, i.e., the algorithm is correct.

Lemma 3. *In the situation described above, suppose that f_i can be maintained with expected generate time T_i, constant update time, space S_i, and initialization time I_i, for $i = 1, \dots, m$, and that g can be maintained with expected generate time T, update time U, space S, and initialization time I. Then the dynamic composition method maintains f with expected generate time $\sum_{i=1}^{m} W_i T_i + T + O(1)$, update time $O(U)$, space $\sum_{i=1}^{m} S_i + S + O(n)$, and initialization time $\sum_{i=1}^{m} I_i + I + O(n)$.*

Proof. The expected time needed is T for the generation according to g, $\sum_{i=1}^{m} W_i T_i$ for the generation according to a restriction of f, and $O(1)$ for miscellaneous overhead. An update to f entails at most two updates to restrictions of f and one update to g, plus whatever updates are caused by changes in the partition $D_1, \dots, D_m$. By assumption, these updates can be executed in $O(U)$ time. □

In attempting to combine Lemmas 2 and 3 directly as sketched above, we are faced with the difficulty that we cannot put a bound on the number of layers, since weights may be arbitrarily small. The weighted average $\sum_{i=1}^{m} W_i T_i$ appearing in Lemma 3, however, shows that we can allow a very bad method (a large T_i) for very small weights (a small W_i). In concrete terms, we can use the naive method of Lemma 1 to handle all weights no larger than $1/n^2$. Since these weights sum to at most $1/n$ and Lemma 1 guarantees a generate time of $O(n)$, the contribution of the small weights to the overall expected generate time will be constant. This means that we need only maintain $m = \lceil 2\log_2 n \rceil$ layers according to Lemma 2; the remaining layers are lumped together and handled by the naive algorithm.

Lemma 4. *A distribution on $D = \{1, \dots, n\}$ can be maintained with constant expected generate time, constant update time, $O(n \log n)$ space, and $O(n \log n)$ initialization time, plus the time and space needed to maintain a distribution on a fixed domain of size $O(\log n)$.*

Proof. We combine Lemmas 1, 2 and 3 as described above. For $i = 1, \dots, m = \lceil 2\log_2 n \rceil$, maintain the restriction f_i of f to $D_{-i} = \{j \in D : f(j) \in R_{-i}\}$ according to Lemma 2, and maintain the restriction f_{m+1} of f to $D_{-m-1} = D \setminus \bigcup_{i=1}^{m} D_{-i}$ according to Lemma 1. Then put the $m+1$ data structures together using Lemma 3; we already argued above that the resulting expected generate time is constant. The space and initialization time needed by each of the $O(\log n)$ data structures is $O(n)$.

The dynamic composition method requires that given $j \in D$, constant time must suffice to determine the unique $i \in \{1, \dots, m+1\}$ with $j \in D_{-i}$. While at first glance this might appear to require the extraction of binary logarithms ($i = \min\{\lceil \log_2(2/f(j)) \rceil, m+1\}$), an appendix argues that in fact we are not going beyond our stated instruction repertoire. □

Lemma 4 still does not provide us with a complete data structure, since we are left with the problem of maintaining a layer distribution on a domain of size $O(\log n)$. Applying Lemma 4 a second time, we can reduce this problem to one of maintaining a distribution on a domain of size $O(\log\log n)$. Continuing in this fashion until the remaining problem is of constant size and hence trivial to solve, we could derive a solution with $O(\log^* n)$ expected generate time and $2^{O(\log^* n)}$ update time. We can do better, however, by exploiting the fact that after a constant number of reduction steps, the problem has become so small that it can be solved by means of table lookup. This is pursued in the next subsection.

2.3 Table Lookup

Our approach will be to approximate the given distribution by a suitable generalized distribution, to store the generalized distribution in a single memory word, and to use this word to index into tables that indicate the outcomes of all possible operations. For this to have any hope of succeeding, the generalized distribution must be defined on a very small domain, which is certainly the case, and its weights must be small integers. As described in the proof of Lemma 6, we achieve this by scaling and rounding the weights of the original distribution. Assume first that this has already taken place.

Lemma 5. *A generalized distribution f on $\{1,\ldots,m\}$ and with integer weights in the range $\{0,\ldots,m-1\}$ can be maintained with constant generate time, constant update time, $m^{O(m)}$ space and initialization time, and a word length of $O(m\log m)$ bits.*

Proof. It suffices to encode f as a single integer F and to realize the following functions in constant time:

Generate: Given F and an integer l from the set $\{1,\ldots,\sum_{j=1}^{m} f(j)\}$ (l will be chosen randomly from this set), return the integer that would be output by the prefix-sums algorithm applied to the generalized distribution f and the random number l, i.e., return the unique $i \in \{1,\ldots,m\}$ with $\sum_{j=1}^{i-1} f(j) < l \leq \sum_{j=1}^{i} f(j)$.

Update: Given F and integers i and j with $1 \leq j \leq m$ and $0 \leq i \leq m-1$ (representing a request to set $f(j)$ to i), return an integer F' representing the generalized distribution f' on $\{1,\ldots,m\}$ with $f'(j) = i$, and $f'(l) = f(l)$ for all $l \in \{1,\ldots,m\}\setminus\{j\}$.

Any reasonable encoding of f will do, e.g., take F as the integer $\sum_{j=1}^{m} f(j)m^{j-1}$ with m-ary representation $f(m)\cdots f(1)$. The functions Generate and Update can be realized via table lookup, whereby the arguments of a function are used as indices into an (up to three-dimensional) array whose entries are the desired answers. The tables have $m^{O(m)}$ entries, and it is easy to compute any given entry in $O(m)$ time. Hence the space and initialization time are $m^{O(m)}$, the necessary word length is $O(m\log m)$ bits, and the generate and update times are constant. □

Before we can apply the above result in the context of the previous subsection, we have to deal with the rounding problem mentioned above.

Lemma 6. *A distribution f on $\{1,\ldots,m\}$ can be maintained with constant expected generate time, constant update time, $m^{O(m^2)}$ space and initialization time, and a word length of $O(m^2\log m)$ bits.*

Proof. Consider the following derived distribution f' on $\{1,\ldots 2m\}$: For $j = 1,\ldots,m$, $f'(j) = (1/m^2)\lfloor m^2 f(j)\rfloor$, and $f'(m+j) = f(j) - f'(j)$. To generate a random value according to f, we can generate a random value according to f' and subtract m if the value generated is larger than m. On the other hand, an update to f translates into a constant number of updates to f'. It hence suffices to maintain f' within the stated bounds.

The values $f'(1),\ldots,f'(m)$ are obviously multiples of $1/m^2$, so that the first half of f' (i.e., the restriction of f' to $\{1,\ldots,m\}$) can be maintained with constant generate and update time and $m^{O(m^2)}$ space and initialization time according to Lemma 5 — maintaining a generalized distribution of integers scaled by a common and fixed factor obviously is as easy as maintaining the generalized distribution of the integers themselves. On the other hand, the values $f'(m+1),\ldots,f'(2m)$ are bounded by $1/m^2$ and therefore sum to at most $1/m$. We can hence maintain the second half of f according to the naive algorithm and use Lemma 3 to combine the data structures for the first and second halves of f, incurring only a constant expected generate time for the second half. □

As outlined at the end of the previous subsection, Lemmas 4 (applied twice) and 6 combine to show that distributions on $\{1,\ldots,n\}$ can be maintained with constant expected generate time, constant update time, and $O(n\log n)$ space and initialization time.

2.4 The Complete Data Structure

In this section we improve the result formulated above by showing how to reduce the space and initialization time to $O(n)$. For this it is necessary to take a second look at the data structures implied by the proof of Lemma 4.

The superlinear resource requirements can be traced to the fact that $m = \Theta(\log n)$ layers of a distribution f are maintained simultaneously using the construction of Lemma 2; for each layer, $\Theta(n)$ space and initialization time are set aside for the arrays A and B. Since layer domains are disjoint, however, all layers can in fact share the same array B — two layers never access the same cell in B. Furthermore, the sizes of those parts of the arrays A actually in use never sum to more than n. We are hence left with the problem of maintaining a collection of dynamic arrays that grow and shrink, but whose total size remains bounded by n. We solve this problem in a way suggested by the known methods for dynamizing static data structures [OvL81].

Our general strategy is as follows: We represent each array by an *array segment*, trying to keep these near the beginning of a large *base segment*, the part of which following the last array segment forms a *free segment*. When an array needs to expand but has no room to do so within its segment, it abandons its segment, which becomes *garbage*, and a new, larger, segment is allocated for it from the beginning of the free segment. Of course, if this were the entire scheme, array segments would tend to migrate towards still higher memory addresses, eating up ever more of the free segment, and no linear bound would apply. To counteract this drift we run a *garbage collector* in parallel with the algorithm. The garbage collector continuously sweeps the base segment and moves the array segments closer to the beginning of the base segment, squeezing out garbage cells as it does so. What is to be shown, basically, is that the garbage collector can keep up with the garbage-producing capability of the algorithm.

Lemma 7. *Suppose that an algorithm maintains a collection $\mathcal{A}$ of dynamic one-dimensional arrays indexed from* 1, *and that in each step of the algorithm a single array in $\mathcal{A}$ may expand or contract by a single memory cell. Then $\mathcal{A}$ can be represented in a fixed contiguous block of $O(n)$ memory cells, where n is a fixed known upper bound on the total size of the arrays in $\mathcal{A}$.*

Proof. Imagine a *base segment* to be laid out from left to right. An array $A \in \mathcal{A}$ of size m is usually stored in an *array segment* of m consecutive cells in the base segment; cells in array segments are said to be *used.* When a segment of initial size m is created for an array A, the m cells to its right are *reserved* for future expansions of A. When A expands, the leftmost cell reserved for it becomes used. When A contracts, the block of cells reserved for it, if nonempty, expands by one cell on the left, but contracts by two cells on the right (i.e., A altogether loses one reserved cell); if no cells were reserved for A before the contraction, this remains so. The cells to the right of the rightmost used or reserved cell are said to be *free* and to form a *free segment*; a *free pointer* to the leftmost free cell is maintained. Cells that are neither used nor reserved nor free are called *garbage cells.*

When an array A of size m needs to expand but has used up all its reserved cells, it abandons its current array segment and moves to a new array segment of initial size $m+1$ allocated for it from the left end of the free segment. In each of the m subsequent accesses to A, a single cell of its old segment is *vacated*, i.e., its contents are copied to the new segment, and it becomes garbage (or free). Since the $m+1$ (additional) cells reserved for A disappear at the rate of at most one per access to A, the old segment of A can be completely vacated before A must move again; hence at any given time A spreads over at most two segments, and it is not difficult to carry out the accesses to A correctly.

The original *user algorithm*, modified as described above, and a *garbage collector* algorithm are run as coroutines: The execution is divided into *rounds*; in each round, the user algorithm is run for one step (in particular, it executes at most one access to an array in $\mathcal{A}$), after which the garbage collector processes 9 cells. The garbage collector needs access to information about the current size and the number of cells reserved for each array segment; we can assume that this information is kept in a cell associated with the first cell of the segment. To understand the workings of the garbage collector, note that we actually employ two base segments, a *foreground segment* and a *background segment*, and that each cell of the foreground segment has an associated *cross pointer*. The garbage collector maintains a *foreground pointer* into the foreground segment and a *background pointer* into the background segment. It always next processes the cell γ pointed to by the foreground pointer. If γ is a garbage cell, processing γ simply increments the foreground pointer. Otherwise the contents of γ are copied to the cell pointed to by the background pointer, the current value of the background pointer is copied to the cross pointer of γ, and the foreground and background pointers are both advanced. The user algorithm normally operates on the foreground segment, but uses the cross pointers to execute any modifications behind the foreground pointer in the background segment as well. Hence when the foreground pointer reaches the free pointer, the information recorded in the two base segments will be identical, except that the background segment may contain less garbage. At this point the roles of the two base segments are switched, the foreground and background pointers are reset to point to the left end of their respective arrays, and the free pointer is set appropriately.

We claim that no sweep of the garbage collector takes more than n rounds, which implies that the total space used is $O(n)$. To see the truth of the claim, assume that it holds for a particular sweep $\mathcal{S}$. Since in each round the user algorithm can produce at most 3 cells of garbage (two by contracting an array, and one by vacating a cell), during $\mathcal{S}$ and the n rounds following $\mathcal{S}$ it produces at most $6n$ cells of garbage. No garbage from before $\mathcal{S}$ can persist after $\mathcal{S}$. Hence at any point during the n rounds following $\mathcal{S}$ there are at most $6n$ cells of garbage. Furthermore, the total number of used and reserved cells at any given time is at most $3n$, for an array of size m occupies at most m used cells in one (new) segment, m used cells in another (old) segment, and m reserved cells. Hence during the n rounds following $\mathcal{S}$ the number of cells that are not free never exceeds $9n$, which implies that the sweep following $\mathcal{S}$ takes at most n rounds. $\square$

Theorem 1. *Distributions on $\{1,\ldots,n\}$ can be maintained with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time.*

3 Maintaining Polynomially-Bounded Generalized Distributions

3.1 Table Lookup Revisited

In this subsection we extend the table lookup method of Lemma 6 from distributions to polynomially-bounded generalized distributions. Recall that the basic idea used in the original scheme was to split each weight in a "significant" part and a small remainder, to maintain the significant parts using table lookup proper, and to maintain the remainders naively. We will use the same approach here, the main obstacle being that the meaning of "significant" changes over time. Because of this, we are actually unable to maintain the remainder weights explicitly.

Lemma 8. *For every fixed $c \geq 0$, a generalized distribution f on $\{1,\ldots,m\}$ and with integer weights bounded by n^c can be maintained with constant expected generate time, constant update time, $O(n + m\log n) + m^{O(m)}$ space and initialization time, and a word length of $O(\log n + m\log m)$ bits.*

Proof. It will be convenient to view each weight as expressed as a sequence of digits in the positional system with base m. Let $t = \lfloor c \log_2 n \rfloor + 1$, a certain upper bound on the maximum number of digits needed. A central data structure will be an $m \times t$ table A that records the most significant part of each weight. Informally, A has a row for each weight and a column for each digit position and stores the three most significant digits of each weight in the three corresponding cells (see Fig. 1).

$f(1)$	$=$	35824					3	5	8		
$f(2)$	$=$	401075				4	0	1			
$\vdots$		67								6	7
		9243380			9	2	4				
$f(m)$	$=$	6921						6	9	2	

Fig. 1. The table A in a decimal example ($m = 10$). Empty squares denote zero entries. The columns of A are numbered from right to left.

More precisely, let us number the rows and columns of A starting from 1 and 0, respectively, and denote the entry in row i and column j of A by $A_{i,j}$, for $i = 1, \ldots, m$ and $j = 0, \ldots, t-1$. For $i = 1, \ldots, m$, if $f(i) = \sum_{j=0}^{l_i} w_{i,j} m^j$ with $w_{i,l_i} \neq 0$ and $w_{i,j} \in \{0, \ldots, m-1\}$, for $j = 0, \ldots, l_i$, then take $A_{i,j} = w_{i,j}$ for all integers j with $\max\{l_i - 2, 0\} \leq j \leq l_i$, and take $A_{i,j} = 0$ for all other j, i.e., for $j \in \{0, \ldots, t-1\} \backslash \{l_i, l_i - 1, l_i - 2\}$.

For $j = 0, \ldots, t-1$, we furthermore record the jth column sum $S_j = \sum_{i=1}^{m} A_{i,j}$ and maintain the generalized distribution f_j on $\{1, \ldots, m\}$ defined by the jth column of A, i.e., $f_j(i) = A_{i,j}$, for $i = 1, \ldots, m$. Since a column holds m integers in the range $\{0, \ldots, m-1\}$, the latter can be done using the simple table-lookup scheme of Lemma 5. Because only three digits of each weight are recorded in A, it is possible to update all this information in constant time whenever a weight changes. We also record the total weight $W = \sum_{i=1}^{m} f(i)$.

Imagine now that we *scale* S_j as well as the entries in the jth column of A by multiplying them by the factor m^j of the corresponding digit position, for $j = 0, \ldots, t-1$. Scaled in this way, the table A corresponds in a natural way to a generalized distribution on a domain of size mt. If we had recorded all digits of each weight in A instead of just three digits, we could have generated a random value z distributed according to f in the following way: First choose a column of A according to the generalized distribution defined by the scaled column weights, then generate z according to the generalized distribution maintained for the chosen column. As it is, however, only the generalized distributions maintained for the leftmost nonzero column of A and the two columns immediately to its right can be trusted.

We actually proceed as follows: Suppose that the index of the leftmost nonzero column of A is $l \geq 2$. Then we use the scaled column weights of columns l, $l-1$ and $l-2$ together with the total weight W to make a 4-way branch in the style of the composition method: To column l, column $l-1$, column $l-2$, or neither of these. In the first three cases we simply proceed to generate a random value according to the generalized distribution maintained for the column in question. In the forth case we spend $\Theta(m)$ time calculating the remainder weights not "covered" by columns l, $l-1$ and $l-2$ and then generate the random value according to these remainder weights using the prefix-sums algorithm of Lemma 1.

The algorithm is easily seen to be correct. It takes constant time, except in the forth case above, which uses $\Theta(m)$ time. Most of the scaled weight is found in the leftmost nonzero column of A or near it, however, and it is not difficult to see that the forth case occurs with probability at most $1/m$. Hence, by Lemma 3, the overall expected generate time is constant.

The above description skirted the question of how to determine the index l of the leftmost nonzero column of A. In order to do this efficiently, we additionally maintain the integer $V = \sum_{j=0}^{t-1} b_j \cdot 2^j$, where $b_j = 1$ if $S_j > 0$ and $b_j = 0$ if $S_j = 0$, for $j = 0, \ldots, t-1$, i.e., V is a bit vector representation of the set of indices of nonzero columns in A. Now l can be obtained as $\lfloor \log_2 V \rfloor$; we again appeal to the description in the appendix of how to compute logarithms. Individual bits in V can be updated using shifts, i.e., multiplications and integer divisions by powers of 2; the need to change a bit is signalled by the transition of the corresponding column sum from zero to nonzero, or vice versa. □

3.2 A Refined Reduction

Recall that the scheme of Lemma 4 maintains a distribution f on $D = \{1, \ldots, n\}$ essentially by taking $m = \lceil 2 \log_2 n \rceil$, maintaining a layer distribution g on the fixed set $I = \{-1, \ldots, -m\}$, maintaining the restriction of f to $D_i = \{j \in D : f(j) \in R_i\}$ for all $i \in I$ using the method of Lemma 2, and maintaining the restriction of f to $D \setminus \bigcup_{i \in I} D_i$ using the naive algorithm. The same approach works when f is a generalized distribution, except that the index set I must be allowed to change over time. Define an integer i to be *nonempty* if $f(j) \in R_i$ for at least one $j \in D$. Then the requirement essentially is that I should contain (at least) the largest nonempty integer and the $m-1$ integers directly below it. In other words, we move a sliding window over the ranges, always focusing on the currently largest weights.

As is clear from the above description, we must make sure that I is slowly dynamic in the sense that an update causes only a constant number of elements to enter or leave I. If this is the case, it is easy to keep the elements of I consecutively numbered (i.e., g can continue to be defined on a fixed set after all) and to carry out the required updates on g. If we furthermore maintain the ith layer of f for all integers i, not just for those currently in I, a *generate* operation can proceed precisely as before.

Let I_0 be the set of all nonempty integers. The set $I' = \{l, l-1, \ldots, l-m+1\}$, where $l = \max I_0$, is probably the most natural choice for I, but I' is not slowly dynamic. We therefore define I not as I', but instead as the set of the $\min\{m, |I_0|\}$ largest elements of I_0. As can easily be verified, this set is indeed slowly dynamic. The only remaining problem is to actually maintain I.

Recall from the end of the previous subsection that a bit vector can be used to maintain an arbitrary set of layer numbers such that the operations *insert*, *delete*, and *findmax* (which returns the maximum) can be executed in constant time. Storing the reversed bit vector as well, we can also execute *findmin* in constant time. We use this to maintain the two sets I and $J = I_0 \setminus I$.

When a new element enters I_0, a comparison with $\min I$ determines whether it is to be inserted in J or in I; if in the latter case subsequently $|I| = m+1$, we also move $\min I$ from I to J. When an element leaves I and $J \neq \emptyset$, we also move $\max J$ from J to I. This maintains I correctly and takes only constant time per update operation.

3.3 The Complete Data Structure

The previous subsection extended Lemma 4 from distributions to polynomially-bounded generalized distributions; furthermore, Lemma 7 still allows us to get by with linear space and initialization time. Applying the reduction twice to a polynomially-bounded generalized distribution on $\{1, \ldots, n\}$, we are left with the problem of maintaining a generalized distribution on a domain of size $O(\log\log n)$ and with weights polynomial in n, at which point we can appeal to Lemma 8.

Theorem 2. *Polynomially-bounded generalized distributions on $\{1, \ldots, n\}$ can be maintained with constant expected generate time, constant update time, $O(n)$ space, and $O(n)$ initialization time.*

4 Maintaining Unrestricted Generalized Distributions

What prevents us from extending the result of the previous section beyond polynomially-bounded generalized distributions is the convention that all integers must be of $O(\log n)$ bits, which restricts the length of the bit vectors used in Subsections 3.1 and 3.2, as well as our inability to compute logarithms of superpolynomial integers in constant time. If these obstacles were removed, i.e., given unit-cost implementations of the arithmetic operations on arbitrary integers and of the function $\text{LOG}_2 : \mathbb{N} \to \mathbb{N} \cup \{0\}$ with $\text{LOG}_2(m) = \lfloor \log_2 m \rfloor$, our scheme could handle arbitrary integer weights. The space used by a straightforward implementation would no longer be bounded, since both the table-lookup method of Lemma 8 and the reduction of Section 3.2 would need an infinite array. Since each weight is recorded in only a constant number of places, however, the distribution data structure could be run in such a way that at any given point in time, only $O(n)$ (known) memory cells would contain essential values. Dynamic perfect hashing could then be employed to pack the $O(n)$ used cells into a fixed storage area of size $O(n)$, i.e., linear space requirements could be achieved. Given the ability to handle arbitrary integer weights, it is easy to handle arbitrary real weights, provided that LOG_2 can be applied to arbitrary positive real numbers: Our data structure allows all current weights to be scaled by a power of 2 in constant time, since all this requires is fixing a new "origin" for the infinite arrays mentioned above. Use this to make sure that all nonzero weights are at least n^2, maintain their integer parts in the data structure for integer weights, and maintain the remainders in the naive way.

A more complicated scheme operating essentially according to these principles was described by Matias, Vitter, and Ni [MVN93]. Their result is that in the stronger model considered above, augmented with the operation $\text{EXP}_2 : \mathbb{N} \to \mathbb{N}$ with $\text{EXP}_2(m) = 2^m$, arbitrary generalized distributions on $\{1, \ldots, n\}$ can be maintained in $O(n)$ space with constant expected generate and update times. In the remainder of this section we sketch a different solution to the same problem in a closely related model. An important parameter in a comparison between the scheme described below and that of Matias, Vitter, and Ni or the one outlined above is the size of the integers manipulated. In all three cases this size is data-dependent and grows to infinity as the maximum ratio between two nonzero weights presented to the data structures grows to infinity. For any fixed sequence of updates, however, the algorithm described below uses integers of roughly n times more bits than its competitors. It also needs the additional operation of bitwise AND on arbitrary nonnegative integers (represented in binary). On the other hand, the algorithm is simple, and it has extremely small constant factors.

Assume first that all weights are integers. The basic idea is to store all prefix sums of the current weights together in a single memory word. Applying the prefix-sums algorithm, we need to locate a random value with respect to the sequence of prefix sums; we do this in constant time using a variant of an essentially parallel algorithm described by Paul and Simon [PS80].

The algorithm manipulates words that conceptually consist of n *fields* of $k+1$ bits each, where k is a positive integer chosen so large that the current total weight fits in k bits. The most significant bit of each field is called its *test bit*, while the remaining k bits form its *main part*. The n fields of each word are packed tightly in its least significant $n(k+1)$ bit positions. For $j = 1, \ldots, n$, define the jth field of a word as its jth least significant field (i.e., the jth field "from the right").

We store a generalized distribution f as the word F whose jth field contains 0 in its test bit and $s_{j-1} = \sum_{i=1}^{j-1} f(i)$ in its main part, for $j = 1, \ldots, n$.

In order to generate a random value z distributed according to f, we begin by drawing a random value l from the uniform distribution on $\{0, \ldots, W-1\}$, where $W = \sum_{j=1}^{n} f(j)$ is the total weight of f. We then create the word L, each of whose n fields contains 1 in its test bit and l in its main part. Observe that if we now subtract F from L, then the test bit in the jth field "survives" if and only if $l \geq s_{j-1}$, for $j = 1, \ldots, n$ (see Fig. 2).

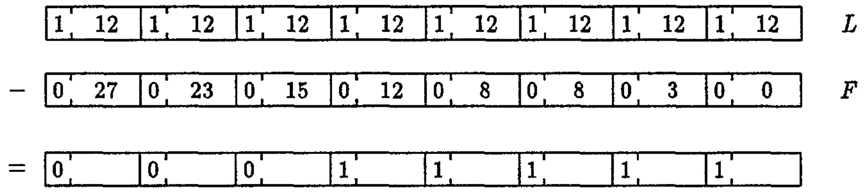

Fig. 2. One subtraction carries out n comparisons.

Hence the number of surviving test bits is precisely the value that we want to output. We can compute this value by masking away all but the test bits, taking the logarithm of the result, adding 1, and dividing by $k+1$.

In order to facilitate the computation of L, we store also the word C, each of whose fields contains 0 in its test bit and 1 in its main part. Then L can be obtained simply as $(l + 2^k) \cdot C$. Assuming that *Random* returns a random value drawn from the uniform distribution on $[0, 1)$, the whole algorithm for generating z hence takes the form

$$z := \frac{\textsc{log}_2\Big(\big((\lfloor W \cdot \mathit{Random} \rfloor + 2^k) \cdot C - F\big) \text{ AND } (2^k \cdot C)\Big) + 1}{k+1}.$$

An update, replacing $f(j)$ by $f'(j)$, say, takes two forms depending on whether the new total weight $W' = W + f'(j) - f(j)$ still fits in k bits. If this is the case, all that is required is to add $i = f'(j) - f(j)$ to all except the j first fields of F; this is realized by the instruction

$$F := F + \left\lfloor \frac{i \cdot C}{2^{j(k+1)}} \right\rfloor \cdot 2^{j(k+1)},$$

which shifts $i \cdot C$ first right and then left in order to clear the j rightmost fields.

If W' does not fit in k bits, it is necessary to increase k, i.e., to "pull apart" the fields in F, after which the update can proceed as above.

We choose the new field size as $k' + 1$ bits, where $k' = \max\{\textsc{log}_2(W') + 1, (n+1)(k+1)\}$, i.e., the new total weight fits in a new field, but in addition each new field can contain at least $n+1$ old fields. Let us call a field of $k+1$ bits a *small field*, and a field of $k'+1$ bits a *large field*.

For all integers $l \geq 2$, let $G(l) = \sum_{j=0}^{n-1} 2^{jl}$; in particular, the new value C' of C is $G(k'+1)$. By the identity $\frac{1}{1-x} = \sum_{j=0}^{\infty} x^j$, for $0 < x < 1$, $2^{nl}/(2^l - 1) = \sum_{j=1}^{\infty} 2^{(n-j)l}$, so that $G(l)$ can be obtained in constant time as $\lfloor 2^{nl}/(2^l - 1) \rfloor$. In order to increase the field size from $k+1$ to $k'+1$, begin by multiplying F by $G(k'-k)$. This creates n copies of the old value of F, but the jth copy "lags behind" the jth large field by $j-1$ small fields, which has the effect of placing the jth old field of F in the leftmost small field of the jth large field, for $j = 1, \ldots, n$; note that the condition $k' \geq (n+1)(k+1)$ ensures that the n copies of F do not overlap. Now simply eliminate all bits outside the leftmost small fields of the large fields using the mask $(2^k - 1) \cdot C'$. Altogether, the expansion of the fields takes the form

$$\begin{aligned} C &:= G(k'+1) \\ F &:= \big(F \cdot G(k'-k)\big) \text{ AND } \big((2^k - 1) \cdot C\big). \end{aligned}$$

The code fragments given above clearly execute in constant time. We hence have

Theorem 3. *In a model that includes constant-time operations on integers of arbitrary size,* LOG$_2$*,* EXP$_2$*, and bitwise* AND*, integer-valued generalized distributions on* $\{1, \ldots, n\}$ *can be maintained with constant generate and update times and* $O(n)$ *space and initialization time.*

The above result is weaker than that of Matias, Vitter, and Ni [MVN93] in that it applies only to integer weights (and in needing the AND operation), but stronger in that the bounds on the generate and update times are worst-case, rather than expected. As argued above, a scheme for arbitrary integer weights can handle arbitrary real weights, provided that it allows all weights to be multiplied by 2^q in constant time, where q is an integer that can be chosen by the algorithm, except that it should be larger than some given q_0. Since this operation essentially reduces to the expansion of fields described above, Theorem 3 holds for arbitrary weights as well, except that "constant generate time" becomes "constant expected generate time".

References

[BFS87] P. Bratley, B. L. Fox, and L. E. Schrage, *A Guide to Simulation (2nd ed.)*, Springer-Verlag, 1987.

[Fox90] B. L. Fox, Generating Markov-chain transitions quickly: I, *Operations Research Society of America Journal on Computing* **2** (1990), pp. 126–135.

[Kle75] L. Kleinrock, *Queueing Systems. Vol. 1: Theory*, John Wiley & Sons, 1975.

[MVN93] Y. Matias, J. S. Vitter, and W. C. Ni, Dynamic generation of discrete random variates, In Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (1993), pp. 361–370.

[OvL81] M. H. Overmars and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, *Information Processing Letters* **12** (1981), pp. 168–173.

[PS80] W. J. Paul and J. Simon, Decision trees and random access machines, In Proc. International Symposium on Logic and Algorithmic, Zürich (1980), pp. 331–340.

[RR91] S. Rajasekaran and K. W. Ross, Fast algorithms for generating discrete random variates with changing distributions, Technical Report No. MS-CIS-91-52, Dept. of CIS, Univ. of Pennsylvania, 1991.

Appendix: Computing Logarithms

We argue that given any fixed $c \in \mathbb{N}$ and $O(n)$ space and initialization time, $\text{LOG}_2(x) = \lfloor \log_2 x \rfloor$ can be computed in constant time for any real $x \in [n^{-c}, n^c]$.

It is easy to set up a table that gives $\text{LOG}_2(x)$ for all $x \in \{1, \ldots, n\}$. For $x \in [1, n^c]$, let $i_0 = \max\{i \geq 0 : \bar{n}^i \leq x\}$, where $\bar{n}$ is the largest power of 2 no larger than n, and then use $\text{LOG}_2(x) = i_0 \cdot \text{LOG}_2(\bar{n}) + \text{LOG}_2(\lfloor x/\bar{n}^{i_0} \rfloor)$. For $x \in [n^{-c}, 1)$, use $\text{LOG}_2(x) \in \{-\text{LOG}_2(1/x), -\text{LOG}_2(1/x) - 1\}$ and choose the correct element of the right-hand set using a precomputed table of powers of 2.