A Linear Algorithm For Generating Random Numbers With a Given Distribution

Michael D. Vose

Abstract— **Let ξ be a random variable over a finite set with an arbitrary probability distribution. In this paper we make improvements to a fast method of generating sample values for ξ in constant time.**

Index Terms—**Random, random-number, random-variable.**

I. Introduction

LET ξ be a random variable distributed over the set $\{a_0, \ldots, a_{n-1}\}$ with corresponding probabilities $\{p_0, \ldots, p_{n-1}\}$. A fast and simple method of generating sample values for ξ has been described by several people (Moss *et al.* [3], Walker [4], Knuth [2]). This method produces a set of sample values in time proportional to sample size. Unfortunately, the method as described requires $O(n \ln n)$ time for initialization. In particular, if the distribution of ξ changes frequently, then the time required to initialize the algorithm to a new distribution becomes a bottleneck. For example, this situation arises in Genetic Algorithms where sample values are needed from a population whose distribution is *constantly* changing [1].

We present a modification which reduces the time required for initialization to $O(n)$. For a simple Genetic Algorithm, this improvement changes an $O(g\, n \ln n)$ algorithm into an $O(g\, n)$ algorithm (where g is the number of generations, and n is the population size). For clarity and completeness we present our version in full detail.

The model of computation we assume includes the following:

- The existence of a constant time uniform random number generator
- a constant time floor operation
- constant time subtraction, comparison, and array reference
- no floating point rounding errors.

This last assumption is partially addressed in Section V, where rounding errors are considered.

II. Specification

The problem is equivalent to producing two algorithms, *init* and *rand*, which share state and satisfy:

- The input to *init*, is an array p representing a probability distribution:

$$p_j \geq 0 \qquad \text{and} \qquad \sum_{j=0}^{n-1} p_j = 1$$

- The effect of *init* is the initialization of *rand* to a function of no arguments (the behavior of *rand* depends only on internal state) which returns an integer j from the set $\{0, \ldots, n-1\}$ with probability p_j.

If the array a contains the range of ξ such that the probability of $\xi = a_j$ is p_j, then a sample value for ξ is obtained by a_{rand}.

III. Algorithms

We assume the existence of the function *uniform*(n) which returns a sample value for a random variable uniformly distributed over the real interval $[0, n)$ in constant time. We also assume the existence of the function $\lfloor \cdot \rfloor$ which returns the floor of its argument in constant time.

A. Rand

Our description of *rand* follows that given by Knuth [2]. Let $prob$ and $alias$ be arrays which are initialized by *init*. The body of *rand* is

u =*uniform*(n)
$j = \lfloor u \rfloor$
If $(u - j) \leq prob_j$ then return
j else return $alias_j$.

Clearly, this algorithm executes in constant time.

B. Init

Our version of *init* proceeds in two stages. The first stage divides the indices of the input into two arrays, $small$ and $large$, via the rule:

$$p_j > 1/n \Rightarrow j \in large$$
$$p_j \leq 1/n \Rightarrow j \in small.$$

The second stage uses the probability distribution p together with $small$ and $large$ to initialize the arrays $prob$ and $alias$. The idea behind this stage is motivated by an analysis of *rand*.

There are two situations in which *rand* returns j:

- If $j = \lfloor u \rfloor$ and $(u - j) \leq prob_j$ then j is returned. This situation occurs with probability

$$\frac{1}{n} prob_j$$

Manuscript received March 8, 1990; revised May 9, 1991. Recommended by M. V. Zelkowitz.

The author is with the Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996.

IEEE Log Number 9102388.

- If $i = \lfloor u \rfloor$, $(u - i) > prob_i$, and $alias_i = j$ then j is returned. This situation occurs with probability

$$\frac{1}{n} \sum_{\substack{i=0 \\ j=alias_i}}^{n-1} 1 - prob_i.$$

First, suppose that $j \in small$, and $prob_j$ were np_j. If every entry of $alias$ is a member of $large$, then only the first situation can occur. Hence *rand* returns j with probability $\frac{1}{n} prob_j = p_j$, as required.

Second, suppose that $k \in large$, and that when the assignment $prob_j = np_j$ was made for the previously considered $j \in small$, the entry $alias_j$ was also defined to be k. Then *rand* could return k with probability $\frac{1}{n}(1 - prob_j)$, which is a term of the second situation. If p_k is then redefined to take this into account via the assignment $p_k = p_k - \frac{1}{n}(1 - prob_j)$, we could iterate these two procedures after reclassifying k as to being $small$ or $large$.

This idea motivates our definition of *init*:

$l = 0 \; ; \; s = 0$
For $j = 0$ to $n - 1$
if $p_j > \frac{1}{n}$
then $large_l = j \; ; \; l = l + 1$
else $small_s = j \; ; \; s = s + 1$
While $s \neq 0$ and $l \neq 0$
$s = s - 1 \; ; \; j = small_s$
$l = l - 1 \; ; \; k = large_l$
$prob_j = n * p_j$
$alias_j = k$
$p_k = p_k + (p_j - \frac{1}{n})$
if $p_k > \frac{1}{n}$
then $large_l = k \; ; \; l = l + 1$
else $small_s = k \; ; \; s = s + 1$
While $s > 0$ do $s = s - 1 \; ; \; prob_{small_s} = 1$
While $l > 0$ do $l = l - 1 \; ; \; prob_{large_l} = 1$.

Clearly, *init* runs in $O(n)$ time. The first loop cycles n times. The second loop decreases $l + s$ on each iteration, and initially $l + s = n$. The last two loops complete this decrement of l and s to 0.

IV. Correctness

The arrays $prob$ and $alias$ produced by *init* are different from those used by the original algorithm. We are therefore obliged to prove the correctness of our solution.

To allow the use of convenient notation, we first establish some conventions.

An array may be regarded as a partial function which maps an index to the corresponding entry. Uninitialized arrays are thought of as having empty domain. If a is an array and $\mathcal{D}$ is its domain, then after an assignment $a_i = \cdots$, the index i is an element of $\mathcal{D}$.

Let $\chi_{\mathcal{D}}$ be the indicator function of the set $\mathcal{D}$ defined by:

$$\chi_{\mathcal{D}}(x) = \begin{cases} 1, & \text{if } x \in \mathcal{D} \\ 0, & \text{otherwise.} \end{cases}$$

An invariant of the first while loop of *init* is that, for all j:

$$\{\chi_{\mathcal{D}}(j)\}\frac{1}{n} prob_j + \frac{1}{n} \sum_{\substack{i=0 \\ j=alias_i}}^{n-1} (1 - prob_i) + \{1 - \chi_{\mathcal{D}}(j)\}p_j = \textit{Probability}[\xi = a_j]$$

where the arrays $prob$ and $alias$ are initially uninitialized and $\mathcal{D}$ is their domain. At entry $\mathcal{D} = \varnothing$, so the invariant becomes:

$$\frac{1}{n} \sum_{\substack{i=0 \\ j=alias_i}}^{n-1} (1 - prob_i) + p_j = \textit{Probability}[\xi = a_j].$$

Note that the sum is empty and hence 0, because the condition $j = alias_i$ is not satisfied when $alias_i$ is undefined. Therefore the invariant holds at entry.

After the body of the while loop has executed, an element j of $small$ has been included in the domain $\mathcal{D}$. Hence the net change to

$$\{\chi_{\mathcal{D}}(j)\}\frac{1}{n} prob_j + \{1 - \chi_{\mathcal{D}}(j)\}p_j$$

is zero since $prob_j = n\,p_j$. Moreover, $small$ and $large$ are kept disjoint, which implies that $j = alias_j$ is not possible. Hence the sum

$$\frac{1}{n} \sum_{\substack{i=0 \\ j=alias_i}}^{n-1} (1 - prob_i)$$

also does not change.

If k is the element of $large$ which was assigned to $alias_j$, then the invariant at k becomes

$$\frac{1}{n} \sum_{\substack{i=0 \\ k=alias_i}}^{n-1} (1 - prob_i) + p_k = \textit{Probability}[\xi = a_k]$$

since the movement of elements is from large to small (if at all), and a precondition for $k \in \mathcal{D}$ is that it was previously in $small$. Note that the new term in this sum corresponds to $i = j$, which represents an increase of

$$\frac{1}{n}(1 - prob_j) = \frac{1}{n} - p_j$$

However, p_k was redefined by $p_k = p_k + p_j - \frac{1}{n}$, which cancels this increase exactly. We have therefore established the first invariant.

Another invariant of the first while loop is that

$$\sum_{j=0}^{s-1} p_{small_j} + \sum_{k=0}^{l-1} p_{large_k} = \frac{1}{n}(s + l).$$

This invariant holds at entry since $s + l = n$, and the probability array p is initially partitioned by $small$ and $large$.

After the body of the while loop has executed, the left-hand side has been decreased by p_j for $j = small_{(s-1)}$, and by $\frac{1}{n} - p_j$ through the assignment $p_k = p_k + p_j - \frac{1}{n}$ for $k = large_{(l-1)}$. Since $s + l$ decreases by 1, the right-hand side also decreases by $\frac{1}{n}$, which establishes the second invariant.

A consequence of this invariant is that the termination condition of the first while loop is equivalent to the single

condition $l = 0$. This follows from the observation that otherwise,

$$\sum_{k=0}^{l-1} p_{large_k} > l\frac{1}{n}$$

which violates the invariant when $s = 0$. Moreover, at termination of the first while loop, we have:

$$0 \le j < s \implies p_{small_j} = \frac{1}{n}$$

since

$$0 \le j < s \implies p_{small_j} \le \frac{1}{n}$$

and when $l = 0$, the invariant is:

$$\sum_{j=0}^{s-1} p_{small_j} = s\frac{1}{n}.$$

If $s = 0$, then neither of the second or third while loops of *init* are entered, and the first invariant reduces to:

$$\frac{1}{n}prob_j + \frac{1}{n}\sum_{\substack{i=0 \\ j=alias_i}}^{n-1}(1 - prob_i) = \textit{Probability}[\xi = a_j]$$

which finishes the proof of correctness for this case.

If $s > 0$, then the second while loop maintains the first invariant, since $p_j = \frac{1}{n}$ for $j = small_{s-1}$ implies that the assignment $prob_j = 1$ leaves

$$\{\chi_{\mathcal{D}}(j)\}\frac{1}{n}prob_j + \{1 - \chi_{\mathcal{D}}(j)\}p_j$$

unchanged. After execution of the second while loop, $s = 0$ and the third while loop is not entered. The proof of correctness is finished as before by appealing to the first invariant.

V. Rounding Errors

The reason for including in *init* the theoretically unnecessary termination condition $s = 0$ and the third while loop which is theoretically never entered is that floating point rounding errors may lead to the misclassification of indices onto $small$ or $large$.

The analysis of the previous section shows that if the first while loop is terminated by $s = 0$, then the remaining elements of $large$ (in positions 0 through $l - 1$) are misclassified. They are therefore treated in an appropriate manner (as if they were in $small$) by the third while loop.

VI. Optimization

In this section we point out some features of our algorithm which, depending on the user's situation, may be exploited to significantly reduce running time.

Subtractive or linear congruential methods for random number generation are fastest when the modulus is $2^{wordsize}$. In some applications a resolution of what typically is 32 bits in the random number generator is not sufficient. In this case, several calls to a 32-bit random integer generator may be used to obtain the required precision. Given this situation, the body of *rand* becomes:

obtain the required number if random bits
$v = (\text{some of the bits}) * constant_1$
$j = \lfloor(\text{the reset of the bits}) * constant_2\rfloor$
If $v \le prob_j$ then return j else return $alias_j$

where $constant_1$ is chosen so that $v \in [0, 1)$, and $constant_2$ is chosen so that $j \in \{0, \ldots, n-1\}$. The reader is cautioned to exercise care in choosing random bits; for example, linear congruential methods yield low-order bits with small cycle times. Note that, according to *init* and *rand*, the comparison $v \le prob_j$ above has the form:

$$(\text{some of the bits}) * constant_1 \le (prob_j = n * p_j)$$

where the assignment takes place in *init*. Therefore redefining $constant_1$ (by dividing it by n) makes the assignment $prob_j = n * p_j$ unnecessary and allows $prob$ and p to be the same array! The appropriate adjustment to the last two while loops (of *init*) is to assign $1/n$ instead of 1.

Further optimizations follow by exploiting a homogeneity property of *init*. Suppose that q is an array such that:

$$p_j = \frac{q_j}{\cdot\sum q_j}$$

Note that

$$p_k = p_k + (p_j - \frac{1}{n}) \iff q_k = q_k + (q_j - \frac{\sum q_j}{n})$$

and

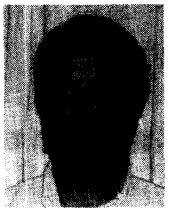
$$p_k > \frac{1}{n} \iff q_k > \frac{\sum q_j}{.n}$$

It follows that if the constant $1/n$ in *init* is replaced by $n^{-1}\sum p_j$, and if $constant_1$ is redefined (multiply it by $\sum p_j$), then the array p need not sum to one! This is very significant because it is almost always faster to compute the direction of a probability vector than it is to determine the actual probabilities.

A final optimization is to eliminate the stacks *small*, *large* and their associated variables s, ℓ which are used by $init$, and hence to also eliminate the initial sorting of indices of p. This is accomplished by letting j and k be indices into p such that p_j would be classified as small (less than $n^{-1}\sum p_j$), and p_k would be classified as large (simply increment j and k until they point at appropriate objects). The details involved (there are a few to consider, and a temporary variable is needed for what was previously the top of *small*) are all straightforward and make an easy exercise for the reader.

References

[1] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley, 1989.
[2] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Reading, MA: Addison-Wesley, 1981, pp. 115–116.
[3] J. K. Moss, R. J. Simpson, and W. Tempest, "A pseudo-random pulse train generator with controllable rate for modeling of audiometric systems," *Radio and Electron. Eng.*, vol. 42, pp. 419–424, 1970.
[4] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Trans. Math Software*, vol. 3, no. 3, pp. 253–256, 1977.



Michael D. Vose was born in San Diego, CA, in 1953. He holds Ph.D. degrees in mathematics (1981) and computer science (1988) from the University of Texas at Austin.

In 1982 he was an Assistant Professor of Mathematics at Texas A&M University. In 1985 he was as Associate Research Scientist at the University of Texas at Austin, and in 1987 he was a member of the technical staff of Computational Logic, Inc. Since 1988 he has been an Assistant Professor of Computer Science at the University of Tennessee, Knoxville.