Sequential Reservoir Sampling with a Non-Uniform Distribution

M. KOLONKO
Technical University of Clausthal

D. WÄSCH
Technical University of Clausthal

Nov. 2004

We present a simple algorithm that allows sampling from a stream of data items without knowing the number of items in advance and without having to store all items in main memory. The sampling distribution may be general, i. e. the probability of selecting a data item i may depend on the individual item. The main advantage of the algorithms is that they have to pass through the data items only once to produce a sample of arbitrary size n.

We give different variants of the algorithm for sampling with and without replacement and analyze their complexity. We generalize earlier results of Knuth on reservoir sampling with a uniform sampling distribution. The general distribution considered here allows to sample an item with a probability equal to the relative weight (or 'fitness') of the data item within the whole set of items.

Applications include heuristic optimization procedures such as genetic algorithms, where solutions are sampled from a population with probability proportional to their fitness.

1. Introduction

In many statistical procedures a random sample of given size n has to be drawn from a set of data items $D := \{d_1, \dots, d_N\}$. If each item has the same chance to be drawn this is the classical problem of random sampling, see [Dev86], chapt. XII, for an overview.

In this paper we present algorithms that are able to take samples when N and the set D are not known in advance. Moreover, the probability for sampling a data item $i \in D$ may depend on its 'weight' $f(i)$ (and on the weights of the items drawn before). Our algorithms need only one pass through the data set, where the items are visited one after another in some given order. They are 'reservoir algorithms' (see [Vit85]) using a list or an array as a 'reservoir' which (after an initialization) always contains a valid random sample of the data items seen so far.

The algorithms may be used if the amount of data is too large to fit into memory and access operations are time consuming or costly, e. g. if the data have to be read from magnetic tapes or remote data bases via Internet. Another application arises in heuristic optimization procedures, where solutions often have to be drawn randomly from a 'population' or 'neighborhood'. Genetic algorithms, e. g., are a popular method for finding the maximum of some reward or 'fitness' function f over a set of solutions. The algorithm runs through a sequence of sets of solutions, so-called populations. Given the $n-$th population, new solutions ('off-spring') are produced from it by genetic operations like mutation or crossover. Often, the $n+1$-st population is then formed by randomly selecting solutions from the off-spring, each with a probability proportional to its fitness relative to the whole set ('survival of the fittest'). Production of off-spring and reduction to the next population are usually two separate steps. Our algorithms allow to perform the selection of the new population already during production without having to store all off-spring. The sequence of new solutions is treated as a stream of data from which the surviving solutions are sampled. This even allows a type of genetic algorithm with just one continuously evolving population.

Our algorithms may also be applied in simulated annealing which is a local search method where new solutions are drawn randomly from the neighborhood of the present solution. Often, this neighborhood is generated successively by applying a set of feasible 'moves' to the present solution and the probability for selecting solution i may depend on a desirable property $f(i)$ of the solution. Our algorithms allow to sample new solutions during the generation of the neighborhood without having to store them in an extra data structure.

In all these situations, our one pass algorithms avoid additional data structures for storing the items and/or extra passes through the data set. The price one has to pay is the generation of additional random numbers. Simulations indicate that even if the data set D fits into memory, our algorithms may be faster than standard algorithms for very large and very small sample sizes n.

For the case of a uniform sampling distribution, [Knu81], p. 138 (see also [Dev86], XII.5) gives a reservoir algorithm that takes one pass through the data set. The first n items are stored in the reservoir which has n positions. Then, the t-th data item visited is taken into the reservoir with probability n/t. The new item is written at a random position, overriding items previously selected. In [Knu69] (p. 123), another version of the reservoir algorithm was given where an auxiliary random number uniformly distributed over the interval $(0,1)$ is drawn for each data item. The sample consists of those items with the n largest auxiliary values.

In [Vit85] and [Li94] these reservoir algorithms are further refined: after the basic algorithm has selected the $t-$th item for the reservoir, it will visit a number S_t of items before it actually selects the next item for the reservoir. The distribution of the random number S_t is easily calculated and its simulation may take less time than than by inspecting all items. Thus, sampling may skip the next S_t items completely and take the S_t+1-st into the reservoir. Depending on the storage medium such a 'fast-forward' operation may increase the speed of the algorithm considerably.

Our algorithms for a general sampling distribution use similar techniques. Algorithm WRS (Weighted Reservoir Sampling) is based on the approach of [Knu69]. We use exponentially distributed auxiliary values with the parameter of the exponential distribution being the weight ('fitness') of the item. Then the largest auxiliary values determine the sample. This algorithm also has a fast-forward version WRS-FF (WRS-Fast-Forward) that is faster than WRS if the sample size n is small compared to the number N of data items. Our fastest algorithm WRS-FFX (WRS-FF Extended) uses algorithm WRS in the beginning and then switches to WRS-FF. For sample size $n = 1$ we are able to give an algorithm WSS (Weighted Single Sampling) similar to [Knu81] which needs no auxiliary values and which may also be used for sampling with replacement.

The paper is organized as follows: In Section 2, we first define precisely what is meant by sampling according to a weight function. We then present our basic algorithm WRS for sampling without replacement and its fast-forward variant WRS-FF in Section 3. Their expected run times are discussed in detail and some empirical results are given in Section 4. The combination WRS-FFX of the two algorithms, which is technically more involved, is presented in Section 5. In the last two sections, special algorithms for samples of size one (Section 6) and for sampling with replacement (Section 7) are given. Some basic facts about exponential distributions are collected in Appendix A.

2. Sampling with a Non-Uniform Distribution

For a simpler presentation, we identify the set of data items with the set $D := \{1, \dots, N\}$ and assume that these items are visited in the order $1, 2, , \dots, N$. Let

$$f : D \to (0, \infty)$$

be a weight function and assume that $f(i)$ can only be calculated after the $i-$th item has been read. The aim is to select item i with a probability proportional to the weight $f(i)$. If we know in advance that $f(i) = \text{const}, i \in D$, then we are in the situation of [Knu69] which we shall refer to as the uniform case.

Let the required sample size be $n \geq 1$.

2.1. Sampling without replacement

Here, we have to assume that $1 \leq n \leq N$. Let

$$\bar{D}_m \ := \ \{(i_1, \dots, i_m) \in D^m \mid i_\mu \neq i_\nu \text{ for all } 1 \leq \nu < \mu \leq m\}, \quad 1 \leq m \leq n \qquad (1)$$

be the set of possible results from sampling m times from D without replacement. Let the required random sample be $(\bar{X}_1, \dots, \bar{X}_n)$. After the selection of the first $1 \leq m < n$ items, the $(m+1)$-st item should be selected according to its weight within the remaining set of items, i. e.

$$\mathbf{P}[\bar{X}_{m+1} = i \mid \bar{X}_1 = i_1, \dots, \bar{X}_m = i_m] := \frac{f(i)}{\sum_{\substack{j=1,\dots,N \\ j \notin \{i_1,\dots,i_m\}}} f(j)} \ = \ \frac{f(i)}{F(N) - \sum_{l=1}^m f(i_l)} \quad (2)$$

for $(i_1, \ldots, i_m, i) \in \bar{D}_{m+1}$ and 0 otherwise. Here we have put $F(t) := \sum_{j=1}^{t} f(j)$. As the first item has the distribution $\mathbf{P}(\bar{X}_1 = i) = f(i)/F(N)$, the joint distribution of $(\bar{X}_1, \ldots, \bar{X}_n)$ is

$$\begin{aligned}
&\mathbf{P}((\bar{X}_1, \ldots, \bar{X}_n) = (i_1, \ldots, i_n)) \\
&\quad = \frac{f(i_1)}{F(N)} \cdot \frac{f(i_2)}{F(N) - f(i_1)} \cdot \frac{f(i_3)}{F(N) - f(i_1) - f(i_2)} \cdot \ldots \cdot \frac{f(i_n)}{F(N) - \sum_{l=1}^{n-1} f(i_l)} \\
&\quad = \prod_{j=1}^{N} \frac{f(i_j)}{F(N) - \sum_{l=1}^{j-1} f(i_l)} \qquad (3)
\end{aligned}$$

for any $(i_1, \ldots, i_n) \in \bar{D}_n$.

2.2. Sampling with Replacement

Let the random sample be $(X_1, \ldots, X_n)$. Sampling with replacement means that the variables are independent each with the same marginal distribution

$$p_i := \frac{f(i)}{\sum_{j=1}^{N} f(j)} = \frac{f(i)}{F(N)}, \quad i \in D. \qquad (4)$$

This distribution gives each item its relative weight within the whole set of data. The resulting joint distribution of the sample is

$$\mathbf{P}((X_1, \ldots, X_n) = (i_1, \ldots, i_n)) = \prod_{j=1}^{n} p_{i_j} = \prod_{j=1}^{n} \frac{f(i_j)}{F(N)} = \frac{f(i_1) \cdot \ldots \cdot f(i_n)}{F(N)^n} \qquad (5)$$

for any $(i_1, \ldots, i_n) \in D^n$.

3. Algorithms for Sampling without Replacement

In contrast to sampling with replacement, the probability (3) for $f \not\equiv \text{const}$ is not invariant under permutations of the sample and the marginal distributions of $\bar{X}_1, \ldots, \bar{X}_n$ are not identical. E. g., for an item i with a relatively large weight $f(i)$, $\mathbf{P}(\bar{X}_1 = i)$ will be larger than $\mathbf{P}(\bar{X}_n = i)$. So the algorithms not only have to decide which items should be in the sample, but also in which order. In particular, the order achieved should be independent of the order in which the items are visited. In our algorithms, all these tasks are solved by the auxiliary values drawn for each item.

The auxiliary value of data item i is an observation of an $\exp(f(i))$–distributed random variable, where $\exp(\alpha)$ is the exponential distribution with parameter α, see also Appendix A. Random numbers v from the $\exp(f(i))$–distribution can be generated by the inversion principle as

$$v := -\log(\texttt{Random()})/f(i), \qquad (6)$$

where `Random()` is a random number generator that produces random numbers according to $U(0,1)$, the uniform distribution on $(0,1)$. Note that these numbers have to be generated independently from each other.

```
ALGORITHM:
(Step 1 takes the first $n$ data items into the reservoir: )
FOR i=1 TO n DO
    read item i and calculate weight f := f(i)
    generate exp(f)-distributed random number v
    create a list element (i,v) and insert it into Res
ENDFOR
(Step2 processes data items $n+1, n+2, \ldots$ : )
    ...
OUTPUT:
Data Items Res[1].i, ..., Res[n].i
```

Figure 1: The Framework for Sampling Without Replacement

3.1. Initialization Step 1

Both algorithms in this section will use an ordered list `Res = (Res[1], ..., Res[n])` as reservoir, where each element can hold an item `Res[m].i` and the auxiliary value `Res[m].v`. The list will always be ordered such that `Res[1].v` $\leq \ldots \leq$`Res[n].v` and will be empty initially. `Res` is an abstract list which will usually be implemented as a heap, where the root contains the item with the maximum auxiliary value. This can be done, as our algorithms mainly need access to the item with the largest auxiliary value in the reservoir. At any time during the execution of the algorithm, `(Res[1].i, ..., Res[n].i)` forms a valid sample from the items seen so far, which we obtain by reading out the heap in ascending order of the auxiliary values. We will come back to this point in Section 4, where we discuss the complexity of the algorithms.

In a common initialization step (Step 1), both algorithms take the first n data items into the reservoir ordered according to their auxiliary values. The algorithms differ in the way they process the remaining $N - n$ items in Step 2, see Figure 1. Here, the function `EOF()` returns `true` if there are no more data items to be read, else `false`.

3.2. Algorithm WRS

After the initialization Step 1, our algorithm WRS continues to observe exponentially distributed auxiliary values for each data item visited with the parameter of the distribution equal to the weight of the item. The reservoir always contains the n items with the smallest auxiliary values seen so far. Figure 2 gives the details of Step 2 for algorithm WRS in the framework of Figure 1.

The following Theorem shows that algorithm WRS produces samples with the distribution as required in (3).

Theorem 3.1. *Let $(\bar{Y}_1, \ldots, \bar{Y}_n)$ denote the items contained in the reservoir list of Algo-*

```
(Step2 :)
WHILE NOT EOF()
    read item i and calculate weight f := f(i)
    generate exp(f)-distributed random number v
    IF v < Res[n].v THEN (take item i into reservoir)
        delete Res[n] (the element with maximal v-value)
        create a list element (i,v) and insert it into Res
    ENDIF
ENDWHILE
```

Figure 2: **Algorithm WRS:** Sampling Without Replacement

rithm WRS after visiting all N data items. Then

$$\mathbf{P}((\bar{Y}_1, \dots, \bar{Y}_n) = (i_1, \dots, i_n)) \;=\; \prod_{j=1}^{n} \frac{f(i_j)}{F(N) - \sum_{l=1}^{j-1} f(i_l)}$$

for all $(i_1, \dots, i_n) \in \bar{D}_n$, i. e. the result of the algorithm has the same distribution as the theoretical sample $(\bar{X}_1, \dots, \bar{X}_n)$ in (3).

For the proof, we use the following simple Lemma on exponentially distributed random variables:

Lemma 3.2. *Let Z_j be an exponentially distributed random variable with parameter $\alpha_j > 0$ for $j = 1, \dots, M$ and assume that $Z_1, \dots, Z_M$ are independent. Then*

$$\mathbf{P}(Z_1 \le Z_2 \le \dots \le Z_M) \;=\; \prod_{j=1}^{M-1} \frac{\alpha_j}{(\alpha_j + \alpha_{j+1} + \dots + \alpha_M)} \tag{7}$$

(For a proof of this Lemma, see Appendix A.)

of Theorem 3.1. Assume that $n < N$, the necessary modifications for the case $n = N$ are easy to see. Recall that we assume to visit the data items $1, \dots, N$ in that order. Let $V_1, \dots, V_N$ be the auxiliary values, i.e. $V_1, \dots, V_N$ are independent random variables, where V_t has distribution $\exp(f(t))$, $1 \le t \le N$. Let $(i_1, \dots, i_n) \in \bar{D}_n$ and put $Z_j := V_{i_j}$ for $1 \le j \le n$ and

$$Z_{n+1} := \min\{V_m \mid 1 \le m \le N, m \notin \{i_1, \dots, i_n\}\}.$$

Note that as $i_1, \dots, i_n$ are fixed, $Z_1, \dots, Z_n, Z_{n+1}$ are independent. Z_j has distribution $\exp(\alpha_j)$, $\alpha_j := f(i_j)$, for $1 \le j \le n$ and from (28) we have that Z_{n+1} again has an exponential distribution with parameter

$$\alpha_{n+1} \;:=\; \sum_{\substack{j=1,\dots,N \\ j \notin \{i_1,\dots,i_n\}}} f(j) \;=\; F(N) - \sum_{l=1}^{n} f(i_l).$$

From the definition of the algorithm we see that

$$\begin{aligned}
&\mathbf{P}(\bar{Y}_1 = i_1, \bar{Y}_2 = i_2, \dots, \bar{Y}_n = i_n) \\
&\quad = \mathbf{P}(V_{i_1} \le V_{i_2} \le \cdots \le V_{i_n} \le \min\{V_m \mid 1 \le m \le N, m \notin \{i_1, \dots, i_n\}\}) \\
&\quad = \mathbf{P}(Z_1 \le Z_2 \le \cdots \le Z_n \le Z_{n+1}).
\end{aligned}$$

With $M := n+1$, the assertion now follows from the Lemma, as we have

$$\frac{\alpha_j}{(\alpha_j + \cdots + \alpha_{M-1} + \alpha_M)} = \frac{f(i_j)}{f(i_j) + \cdots + f(i_n) + F(N) - \sum_{l=1}^{n} f(i_l)} = \frac{f(i_j)}{F(N) - \sum_{l=1}^{j-1} f(i_l)}. \qquad (8)$$

□

Note, that if we have $f \equiv$ const, then (6) shows that the n smallest values of the auxiliary exp(const)−distributed values are exactly those for which the n largest uniform random numbers have been generated. Hence, in this case, our algorithm produces the same sample as the reservoir algorithm of [Knu69]. The additional evaluations of the log-function in (6) are the price for not knowing $f \equiv$ const in advance.

3.3. An Algorithm with 'Fast-Forward'

As was mentioned in the Introduction, the sampling in the uniform case can be accelerated considerably if the number S_t of items not taken into the reservoir is simulated and if these items are simply skipped. In our general case we cannot skip items entirely, as we have to know their weights to obtain the distribution (3). Nevertheless we can save some time if we do not have to draw random numbers for each item.

First, we have to analyze the stochastic behavior of algorithm WRS more closely. We assume that the stream of data items is infinite. Let $(V_t)_{t\ge1}$ be a sequence of independent random variables where V_t has distribution $\exp(f(t))$ as in the proof of Theorem 3.1. $(V_t)_{t\ge1}$ may be looked at as a stochastic process with discrete 'time' t. To keep track of the reservoir contents we have to observe this process at those times where an item is selected. To give a formal definition, let $x_{[\,]} = (x_{[1]}, \dots, x_{[m]})$ denote the vector of values from $x = (x_1, \dots, x_m)$ in ascending order. Then $x_{[n]}$ is the largest of the n smallest values of x. Let $\mathbb{N}$ denote the positive integers.

In algorithm WRS, a new item is put into the reservoir each time V_t is smaller than the threshold represented by the present largest value in the reservoir. After the initialization Step 1 in Figure 1, the reservoir contains the auxiliary values $V_1, \dots, V_n$. With the definition given above, $R_0 := (V_1, \dots, V_n)_{[n]}$ is the threshold after initialization. Let

$$S(m, v) \ := \ \min\{k > 0 \mid V_{m+k} < v\}, \quad m \in \mathbb{N}, v > 0, \qquad (9)$$

then $S(m, v) - 1$ is the number of items skipped after the $m-$th item is processed, if the threshold of the reservoir is v at that time. Define $T_0 := n$, and

$$\begin{aligned}
T_{l+1} \ &:= \ T_l + S(T_l, R_l), \\
R_{l+1} \ &:= \ (V_1, \dots, V_n, V_{T_1}, \dots, V_{T_{l+1}})_{[n]}
\end{aligned} \qquad (10)$$

for integers $l \geq 0$. Then T_l is the $l-$th item selected for the reservoir in Step 2 of algorithm WRS and R_l is the largest auxiliary element in the reservoir after T_l has been processed. Let

$$\tau(N) := \max\{l \in \mathbb{N} \mid T_l \leq N\}$$

be the total number of selections in Step 2. Then the final sample consists of the items belonging to the n smallest auxiliary values of

$$(V_1, \ldots, V_n, V_{T_1}, \ldots, V_{T_{\tau(N)}}).$$

To simulate the selection behavior of algorithm WRS in Step 2 for a fast-forward operation it is therefore sufficient to simulate the embedded stochastic process

$$W_l := (T_l, V_{T_l}), \quad l = 0, 1, 2, \ldots \tag{11}$$

for a fixed initialization $(V_1, \ldots, V_n) = (v_1, \ldots, v_n)$. The next Theorem gives the distribution of this process and shows how to simulate it.

Theorem 3.3. *For all $l \geq 0$, W_{l+1} is independent from $V_1, \ldots, V_n, W_0, \ldots, W_{l-1}$ given T_l, R_l. The exact transition probability of the process is given by*

$$\begin{aligned}
&\mathbf{P}\Big(T_{l+1} = k,\ V_{T_{l+1}} \leq u \ \mid \ T_1 = m_1, \ldots, T_{l-1} = m_{l-1}, T_l = m_l, \\
&\qquad\qquad V_1 = v_1, \ldots, V_n = v_n, V_{T_1} = u_1, \ldots, V_{T_l} = u_l\Big) \\
&= \mathbf{P}\Big(T_{l+1} = k,\ V_{T_{l+1}} \leq u \ \mid T_l = m_l, R_l = r\Big) \\
&= \mathbf{P}\Big(V_k \leq u \mid V_k \leq r\Big) \cdot \mathbf{P}\Big(S(m_l, r) = k - m_l\Big)
\end{aligned}$$

for R_l as defined in (10)*, $r := (v_1, \ldots, v_n, u_1, \ldots, u_l)_{[n]}$, and $0 \leq m_l \leq k$. Moreover*

$$\mathbf{P}(S(m_l, r) \leq k) \ = \ 1 - \mathrm{e}^{-r \sum_{j=1}^{k} f(m_l + j)} \tag{12}$$

and

$$\mathbf{P}(V_k \leq x \mid V_k \leq r) \ = \ \frac{1 - \mathrm{e}^{-x f(k)}}{1 - \mathrm{e}^{-r f(k)}} \tag{13}$$

for $0 \leq x \leq r$ and $k \in \mathbb{N}$.

Proof. With the abbreveations

$$\begin{aligned}
&\bar{T}_l := (T_1, \ldots, T_l), \quad \bar{m} = (m_1, \ldots, m_l) \quad \text{and} \\
&\bar{V}_l := (V_1, \ldots, V_n, V_{T_1}, \ldots, V_{T_l}), \quad \bar{u} := (v_1, \ldots, v_n, u_1, \ldots, u_l)
\end{aligned}$$

we obtain

$$\begin{aligned}
&\mathbf{P}\Big(T_{l+1} = k, V_{T_{l+1}} \leq u \mid \ \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}\Big) \qquad (14) \\
&\quad = \quad \mathbf{P}\Big(V_{T_{l+1}} \leq u \mid T_{l+1} = k, \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}\Big) \cdot \mathbf{P}\Big(T_{l+1} = k \mid \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}\Big).
\end{aligned}$$

With $r := (v_1, \ldots, v_n, u_1, \ldots, u_l)_{[n]},\ m_l \le k$ the condition in (14) fulfills

$$\begin{aligned}
&T_{l+1} = k, \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u} \\
\Longleftrightarrow\ &T_{l+1} = k, \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}, R_l = r \\
\Longleftrightarrow\ &V_{m_l+1} \ge r, V_{m_l+2} \ge r, \ldots, V_{k-1} \ge r, V_k < r, \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}, R_l = r
\end{aligned} \tag{15}$$

From (15) and the independence of the V_m we therefore obtain

$$\mathbf{P}\Big(V_{T_{l+1}} \le u \ \mid T_{l+1} = k, \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}\Big) = \mathbf{P}\Big(V_k \le u \ \mid \ V_k < r\Big). \tag{16}$$

Similarly, for the last expression in (14)

$$\begin{aligned}
\mathbf{P}\Big(T_{l+1} = k \mid \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}\Big) \ &= \ \mathbf{P}\Big(T_{l+1} = k \mid \bar{T}_l = \bar{m}, \bar{V}_l = \bar{u}, R_l = r\Big) \\
= \mathbf{P}\Big(V_{m_l+1} \ge r, V_{m_l+2} \ge r, \ldots, &V_{k-1} \ge r, V_k < r \mid T_l = m_l, R_l = r\Big) \\
= \mathbf{P}\Big(V_{m_l+1} \ge r, V_{m_l+2} \ge r, \ldots, &V_{k-1} \ge r, V_k < r\Big) \\
= \mathbf{P}\Big(S(m_l, r) = k - m_l\Big).&
\end{aligned}$$

We have further

$$\begin{aligned}
\mathbf{P}\Big(S(m,r) > k\Big) \ &= \ \mathbf{P}\Big(V_{m+1} \ge r, V_{m+2} \ge r, \ldots, V_{m+k} \ge r\Big) \\
&= \ \mathrm{e}^{-rf(m+1)} \cdot \mathrm{e}^{-rf(m+2)} \cdot \ldots \cdot \mathrm{e}^{-rf(m+k)} \\
&= \ \mathrm{e}^{-r\sum_{j=1}^{k} f(m+j)}
\end{aligned}$$

proving (12) and for $x \le r$

$$\mathbf{P}(V_k \le x \mid V_k \le r) \ = \ \frac{\mathbf{P}(V_k \le x)}{\mathbf{P}(V_k \le r)} = \frac{1 - \mathrm{e}^{-xf(k)}}{1 - \mathrm{e}^{-rf(k)}}$$

□

To perform a 'fast-forward', we have to simulate the conditional distribution of $(T_{l+1}, V_{T_{l+1}})$ given the contents of the reservoir after T_l has been inserted. Theorem 3.3 shows that this is the conditional distribution of $(T_{l+1}, V_{T_{l+1}})$ given $(T_l, R_l) = (m, r)$. We may then proceed as follows: We fist simulate $S(m,r)$ with the distribution given by (12). This can be done by inversion, where we add the weights $f(m+1), f(m+2), \ldots, f(m+k)$ until

$$\sum_{j=1}^{k} f(m+j) \ \ge -\log(\texttt{Random()})/r \tag{17}$$

for the first time. Note that the right-hand side of (17) is an $\exp(r)-$distributed random variable. After we obtained $k = S(m,r)$ item $m+k$ is taken into the reservoir and we

```
(Step 2:)
put r:=Res[n].v
generate exp(r)-distributed random number y and set F := 0
WHILE NOT EOF()
    read item i and calculate weight f := f(i)
    F := F + f
    IF F > y THEN (take item $i$ into the reservoir)
        v := -log(1-Random()·(1-exp(-f·r)))/f (the auxiliary value, see (13))
        delete Res[n] (the element with maximal v-value)
        create a list element (i,v) and insert it into Res
        put r:=Res[n].v
        generate exp(r)-distributed random number y and set F := 0
    ENDIF
ENDWHILE
```

Figure 3: **Algorithm WRS-FF:** Step 2 of the Fast-Forward Variant

have to draw its auxiliary value which must have a conditional distribution as in (13). This is obtained via inversion by

$$\frac{-\log\left(1 - \texttt{Random()} \cdot (1 - \mathrm{e}^{-r \cdot f(m+k)})\right)}{f(m+k)}. \tag{18}$$

We call this algorithm WRS-FF. It starts with the initialization Step 1 as in Figure 1, the formal definition of its Step 2 is given in Figure 3.

4. Expected Runtimes of Algorithms WRS and WRS-FF

As in [Vit85] Theorem 1, it it clear, that every algorithm for this sampling problem is a type of reservoir algorithm. For the uniform case, i. e. if $f \equiv$ const is known, Vitter showed that if the time for reading (and skipping) data items is ignored, then $O(n(1+\log(N/n)))$, the expected number of selections, is a lower bound for the expected runtime of reservoir algorithms. It is attained for the fast-forward algorithms in [Vit85] and [Li94].

To produce a sample with distribution as given in (3), any algorithm has to visit each item $i = 1, \ldots, N$ individually, calculate its weight and determine whether the item is taken into the reservoir, which is done by a comparison of two reals in our algorithms. This is necessary for all N items, for otherwise, if the algorithm would operate only on a subset of the data set, then this subset itself must be a representative sample, but this is just what the algorithm has to produce.

Hence, a linear effort $O(N)$ is common to all algorithms. For a more subtle study of the expected runtimes we shall therefore drop this common term from our calculation.

For a more subtle study of the expected runtimes we shall therefore drop this common term from our calculations.

For the complexity of keeping the reservoir up to date, we may argue as in [Vit85]: the $t-$th item must become a member of the reservoir with a probability q_t, to be calculated below. Hence the expected number of necessary selections is $\sum_{t=1}^{N} q_t$. As was pointed out before, the marginal distributions of the theoretical sample $(\bar{X}_1, \ldots, \bar{X}_n)$ from (3) are neither identical nor uniform. Hence, a sampling algorithm also has to insert the item at a suitable position of the sample. This needs at least time $O(\log m)$ on the average, where m is the length of the present sample. In the uniform case, this step is implicitly performed, as any random permutation of the sample has the same distribution.

Neglecting the linear effort as mentioned above, this yields a lower bound for the average time complexity of a general reservoir algorithm of

$$O\Big(\sum_{t=1}^{n} q_t \log t \; + \sum_{t=n+1}^{N} q_t \log n\Big). \tag{19}$$

To determine q_t, the probability that the $t-$th item enters the reservoir, we may assume that all auxiliary random values $(v_1, \ldots, v_N)$ produced by the algorithm are different from each other. If we do not have any information about the weight function f, then all permutations of $(v_1, \ldots, v_N)$ are equally likely to appear. Therefore we have

$$q_t \quad = \quad \begin{cases} 1 & \text{for } t \le n \\ \frac{\sum_{k=1}^{n}(t-1)!}{t!} & \text{for } t > n \end{cases} \quad = \quad \begin{cases} 1 & \text{for } t \le n \\ \frac{n}{t} & \text{for } t > n \end{cases} \tag{20}$$

Here, $\sum_{k=1}^{n}(t-1)!$ is the number of possible permutations of $(v_1, \ldots, v_t)$ such that the t-th entry is among the n smallest. Note that the probability q_t for inserting item t into the reservoir when nothing is known about the weight function is the same as in the uniform case, see [Knu81]. We use

$$\sum_{t=n+1}^{N} q_t \;=\; \sum_{t=n+1}^{N} \frac{n}{t} \;\approx\; n \log \frac{N}{n}. \tag{21}$$

Thus (19) becomes

$$O\left(\log n! \; + \; n \log n \log \frac{N}{n}\right) \;\subset\; O\left(n \log n \Big(1 + \log \frac{N}{n}\Big)\right). \tag{22}$$

Compared to the lower bound $O(n(1 + \log(N/n)))$ for the uniform case (see [Vit85]) we have the additional factor $\log n$ caused by the sorting.

4.1. The Expected Runtime of Algorithm WRS

As mentioned before, we assume that the ordered list `Res[1],...,Res[m]` is implemented as a heap. Then operations 'insert' and 'delete the largest element' each take time $c_{\mathrm{hp}} \log m$, where m is the length of the list and c_{hp} is the time constant for a single update step in the heap. Let c_{exp} be the time necessary to produce an exponentially distributed number.

For Step 1 in Figure 1 we obtain an expected runtime

$$T_1(n) \;\approx\; \sum_{t=1}^{n}(c_{\text{exp}} + c_{\text{hp}} \cdot \log t) \;=\; nc_{\text{exp}} + c_{\text{hp}} \cdot \log n! \tag{23}$$

which is in $O(n \log n)$.

In Step 2 algorithm WRS needs to generate one random number for each data item. For each update of the reservoir it needs one insert and one delete-operation. Finally, the sample has to be extracted from the heap taking time $\sum_{m=1}^{n} c_{\text{hp}} \log m$. Therefore we obtain as expected runtime

$$\begin{aligned} T_{\text{WRS}}(N,n) &\approx\; T_1(n) + (N-n)c_{\text{exp}} + \sum_{t=n+1}^{N} q_t \cdot 2c_{\text{hp}} \cdot \log n + c_{\text{hp}} \sum_{m=1}^{n} \log m \\ &\approx N c_{\text{exp}} + c_{\text{hp}} \left(2 \log n! + 2n \log n \cdot \log \frac{N}{n} \right) \end{aligned} \tag{24}$$

which is in $O\Big(N + \log n! + n \log n \log \frac{N}{n}\Big)$.

4.2. Expected Runtime of Algorithm WRS-FF

As above, algorithm WRS-FF needs time $T_1(n)$ to process the first n items. Then in Step 2, for each inserted item two random-numbers have to be generated: the 'skip-value' y on the right hand side of (17), which is $\exp(r)$-distributed and takes time c_{exp} and the conditional auxiliary value v from (18) which takes time c_{cond}. Here we neglect the addition of the fitness-values in (17). Again, the final sample has to be extracted from the heap. As the probability to select an item in WRS-FF is the same as in WRS, we obtain

$$\begin{aligned} T_{\text{WRS-FF}}(N,n) &\approx\; T_1(n) + \sum_{t=n+1}^{N} q_t(c_{\text{exp}} + c_{\text{cond}} + 2c_{\text{hp}} \log n) + c_{\text{hp}} \sum_{m=1}^{n} \log m \\ &\approx nc_{\text{exp}} \left(1 + \log \frac{N}{n}\right) + c_{\text{hp}} \left(2 \log n! + 2n \log n \log \frac{N}{n}\right) + c_{\text{cond}} n \log \frac{N}{n} \end{aligned} \tag{25}$$

Therefore the expected runtime of WRS-FF lies in

$$O\Big(\log n! + n \log n \log \frac{N}{n}\Big).$$

which is the lower bound from (22). A detailed comparison between T_{WRS} and $T_{\text{WRS-FF}}$ shows that approximately

$$T_{\text{WRS}}(N,n) \;>\; T_{\text{WRS-FF}}(N,n) \quad\Longleftrightarrow\quad \frac{\frac{N}{n} - 1}{\log \frac{N}{n}} \;>\; 1 + \frac{c_{\text{cond}}}{c_{\text{exp}}} \tag{26}$$

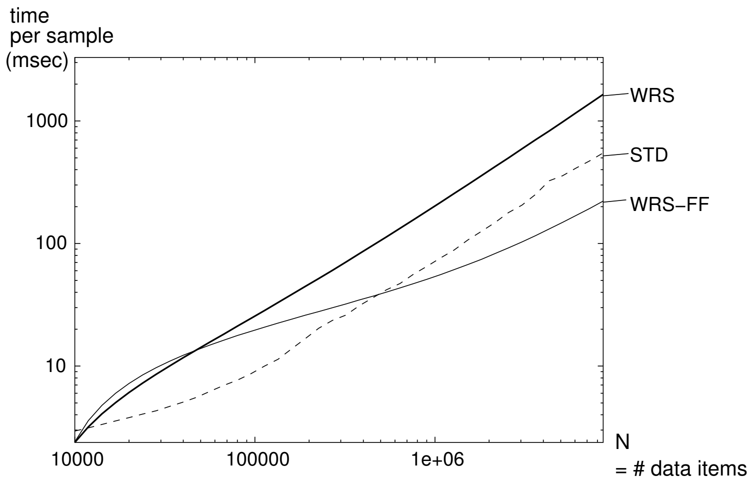

Figure 4: Runtimes of the Algorithms WRS, WRS-FF and STD as Function of N

As can be seen from (26) and the empirical results below, WRS-FF is superior to WRS only for certain parameter values. This will be used in the hybrid algorithm WRS-FFX in section 5.

4.3. Empirical Comparison of Algorithms WRS and WRS-FF

To get an idea of how fast our algorithms really are, we have performed simulation runs. We have compared WRS and WRS-FF to a two-pass standard algorithm STD that needs to read the data into memory. It turned out that even in this situation one of our algorithms was considerably faster than STD for $n \ll N$.

The algorithm STD passes once through the data, reads it into memory, collects information about the weights and then performs n selections using inversion to simulate the sampling probabilities (2). To improve the runtime of STD we used a binary tree for the search process during inversion as recommended by [Dev86], sec. III.2.3. Note that STD is a sophisticated version of the 'roulette-wheel selection' sometimes recommended for genetic algorithms, e. g. [Ree03] p. 65.

Figure 4 shows the average runtimes (in milliseconds) per sample for a fixed sample size $n = 10\,000$ as a function of $N \geq n$. Note that we used a logarithmic scale on both axes. All values are averages over 100 different weight functions, each sampled 100 times. The weights $f(i), i = 1, \ldots, N$ were taken uniformly distributed over $(0, 1)$.

In the experiments of Figure 4, WRS-FF was faster than STD for large values of N, i. e. for $n \ll N$. For $n \approx N$, both WRS and WRS-FF were faster than STD. Also WRS is better than WRS-FF for small values of N as was to be expected from (26). In our implementation, $T_{\mathrm{WRS}}(N, n)$ is smaller than $T_{\mathrm{WRS-FF}}(N, n)$ for $N \leq 4.7n$ (approximately), which means by (26) $c_{\mathrm{cond}} \approx 1.4 c_{\mathrm{exp}}$.

Similar pictures were obtained for other values of n and other types of randomly produced weights (e. g. normally distributed or discrete, see also Figure 5).

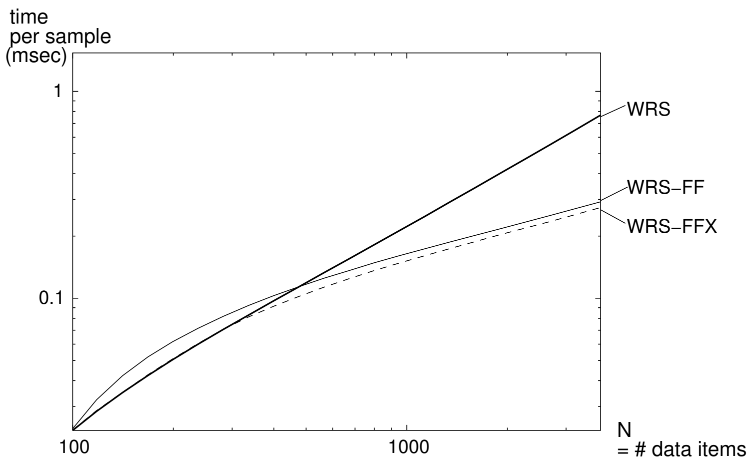

Figure 5: Runtimes of the Algorithms WRS, WRS-FF and WRS-FFX as Function of N

Note that STD needs much more space than WRS and WRS-FF: STD needs space for the N data items and for $2N$ real values for the binary search tree, whereas our algorithms only need space for n data items and n auxiliary values.

All algorithms have been implemented in C++, using GCC 3.2.3 on an AMD Opteron 244 under Debian Linux with kernel version 2.6.3.

5. Switching Between Algorithms WRS and WRS-FF

Figure 4 suggests a combination of the two algorithms WRS and WRS-FF which is calles WRS-FFX. It processes the first $k > n$ items by algorithm WRS, then to use algorithm WRS-FF for the remaining $N - k$ items. To find an optimal value k, we first calculate the expected runtime of WRS-FFX.

$$\begin{aligned} T_{\mathrm{WRS-FFX}}(N, n, k) &= T_{\mathrm{WRS}}(k, n) + \sum_{t=k+1}^{N} q_t(c_{\mathrm{exp}} + c_{\mathrm{cond}} + 2c_{\mathrm{hp}} \log n) \\ &= 2c_{\mathrm{hp}} \log n! + kc_{\mathrm{exp}} + n \sum_{t=k+1}^{N} \frac{1}{t}(c_{\mathrm{exp}} + c_{\mathrm{cond}}) + 2c_{\mathrm{hp}} n \log n \sum_{t=n+1}^{N} \frac{1}{t} \end{aligned}$$

$T_{\mathrm{WRS-FFX}}(N, n, k)$ is minimized for k with

$$k \le n\left(1 + \frac{c_{\mathrm{cond}}}{c_{\mathrm{exp}}}\right) < k + 1.$$

As this expression is actually independent of N and the weight function, we may use it as parameter of the algorithm WRS-FFX. In our implementation $c_{\mathrm{cond}} \approx 1.4c_{\mathrm{exp}}$ and therefore $k \approx 2.4n$ is the right level for switching.

ALGORITHM:

```
F := 0
WHILE NOT EOF()
    read item i and calculate weight f := f(i)
    F := F + f
    IF F·Random() < f THEN Res = i
ENDWHILE
```

OUTPUT:

Data item `Res`

Figure 6: **Algorithm WSS**: Sampling One Element

Figure 5 shows simulation results with this value of k. Here, the sample size is fixed to $n = 100$ and the Figure shows the average time per sample as function of the size N of the data set. This time, the weights are uniformly distributed over $\{1, 2, \dots, 10\}$, while the other parameters are as in Figure 4. Note that WRS-FFX is identical to WRS for $N \leq k$. Algorithm WRS-FFX is clearly at least as fast as WRS and WRS-FF for all values of N.

6. Drawing Samples of Size One

If only $n = 1$ item has to be drawn, the initialization Step 1 in Figure 1 can be skipped and the sorted list `Res` is just a variable. After the $(t-1)-$th item has been processed, the reservoir contains the item with auxiliary value $\min\{V_1, \dots, V_{t-1}\}$. The probability that the $t-$th item goes into the reservoir is therefore

$$\mathbf{P}(V_t < \min\{V_1, \dots, V_{t-1}\}) \;=\; \frac{f(t)}{F(t)}$$

(see Lemma 3.2 and Eq. (28) below). Hence we may simplify algorithm WRS by selecting the $t-$th item with that probability. This yields algorithm WSS (Weighted Single Selection) given in Figure 6. It is more in the spirit of [Knu81] and has a better expected runtime than $T_{\mathrm{WRS}}(N, 1)$, as the constant $c_{\exp}$ is replaced by the smaller constant for a call to `Random()`.

The following Theorem shows that WSS produces the right distribution.

Theorem 6.1. *Let Y be the item in `Res` after all items have been visited by algorithm WSS. Then*

$$\mathbf{P}(Y = i) \;=\; \frac{f(i)}{F(N)}, \qquad \textit{for } i = 1, \dots, N$$

as required by (3) *for $n = 1$.*

Proof. Let $Z_i := 1$ if item i is selected and $Z_i = 0$ otherwise for $i = 1, \ldots, N$. Then $Z_1, \ldots, Z_N$ are independent with $\mathbf{P}(Z_i = 1) = f(i)/F(i)$ and $\mathbf{P}(Z_i = 0) = F(i-1)/F(i)$. As `Res` contains the last item selected, we have for $i \in \{1, \ldots, N\}$

$$\begin{aligned}\mathbf{P}(Y = i) &= \mathbf{P}(Z_i = 1, Z_{i+1} = 0, \ldots, Z_N = 0) \\ &= \frac{f(i)}{F(i))} \frac{F(i)}{F(i+1)} \cdot \ldots \cdot \frac{F(N-1)}{F(N)} = \frac{f(i)}{F(N)}\end{aligned}$$

□

The fast-forward idea may also be applied to algorithm WSS. If the $t-$th item has been selected for the reservoir, the next item to be selected is given by the random variable

$$S(t) := \min\left\{k \in \mathbb{N} \mid U_k < \frac{f(t+k)}{F(t+k)}\right\} \tag{27}$$

where $U_1, \ldots, U_N$ are i.i.d. and $U(0,1)-$distributed.

Theorem 6.2. *For $t, k \in \mathbb{N}$ we have*

$$\mathbf{P}(S(t) \leq k) = 1 - \frac{F(t)}{F(t+k)}$$

and $S(t)$ may be simulated by

$$\min\{l \in \mathbb{N} \mid F(t+l) \geq F(t)/U\}$$

where U is $U(0,1)-$distributed.

Proof. We have

$$\begin{aligned}\mathbf{P}(S(t) > k) = \mathbf{P}(&\frac{f(t+1)}{F(t+1)} \leq U_1, \ldots, \frac{f(t+k)}{F(t+k)} \leq U_k) \\ &= \frac{F(t)}{F(t+1)} \cdot \ldots \cdot \frac{F(t+k-1)}{F(t+k)} = \frac{F(t)}{F(t+k)}.\end{aligned}$$

The second assertion follows form the inversion formula and the fact that U and $1-U$ have the same distribution. □

Figure 7 shows the algorithm WSS-FF. It uses only $\log N$ calls to `Random()` in expectation. Therefore WSS-FF is faster than WSS for all $N \geq 1$.

7. A Reservoir Algorithm for Sampling with Replacement

For sampling *with* replacement we may use an algorithm for sample size 1, e. g. WRS-FF, and run n instances in parallel. This yields algorithm WRS-WR (WRS With Replacement) in Figure 8. It uses an array `(Res[1], ..., Res[n])` of data items as reservoir and an array `(Ft[1], ..., Ft[n])` for real numbers. Note that the first item is taken into all places of the reservoir. Apart from reading the items and evaluating their weights, the algorithm needs $n \log N$ calls to the random number generator in expectation.

ALGORITHM:

```
F := 0; Ft := 0
WHILE NOT EOF()
    read item i and calculate weight f := f(i)
    F := F + f
    IF F $\geq$ Ft THEN
        Res := i
        u := Random(); Ft := F/u
    ENDIF
ENDWHILE
```

OUTPUT:

Data item Res

Figure 7: **Algorithm WSS-FF**: Sampling One Element with Fast-Forward

ALGORITHM:

```
F := 0;
FOR m = 1 TO n DO
    Ft[m] := 0
ENDFOR
WHILE NOT EOF()
    read item i and calculate weight f := f(i)
    F := F + f
    FOR m = 1 TO n DO
        IF F $\geq$ Ft[m] THEN
            Res[m] := i
            u := Random(); Ft[m] := F/u;
        ENDIF
    ENDFOR
ENDWHILE
```

OUTPUT:

Data items Res[1], ..., Res[n]

Figure 8: **Algorithm WRS-WR**: Fast-Forward Sampling with Replacement

APPENDIX

A. EXPONENTIALLY DISTRIBUTED RANDOM VARIABLES

We recall some simple facts from elementary stochastics. The exponential distribution $\exp(\alpha)$ with parameter $\alpha > 0$ has density $t \mapsto \alpha e^{-\alpha t}$ and distribution function $1 - e^{-\alpha t}$ for $t \geq 0$. If $Z_1, \dots, Z_m$ are independent random variables where Z_t is $\exp(\alpha_t)$ distributed, $t = 1, \dots, m$, then $\min\{Z_1, \dots, Z_m\}$ has distribution

$$\exp(\alpha_1 + \cdots + \alpha_m). \tag{28}$$

of Lemma 3.2. We first show by downward induction on $l = M, M-1, \dots, 1$ that for $t \geq 0$

$$\mathbf{P}(t \leq Z_l \leq Z_{l+1} \leq \cdots \leq Z_M) \;=\; \prod_{j=l}^{M-1} \frac{\alpha_j}{(\alpha_j + \alpha_{j+1} + \cdots + \alpha_M)} \mathrm{e}^{-(\alpha_l + \cdots + \alpha_M)t}. \tag{29}$$

For $l = M$ we have $\mathbf{P}(t \leq Z_M) = e^{-t\alpha_M}$. As Z_{l-1} is independent from $Z_l, \dots, Z_M$, we obtain for the induction step

$$\begin{aligned}
&\mathbf{P}(t \leq Z_{l-1} \leq Z_l \leq \cdots \leq Z_M) \\
&\quad = \int_t^\infty \mathbf{P}_{Z_{l-1}}(ds)\ \mathbf{P}(s \leq Z_l \leq Z_{l+1} \leq \cdots \leq Z_M) \\
&\quad = \int_t^\infty \alpha_{l-1} \mathrm{e}^{-s\alpha_{l-1}} \prod_{j=l}^{M-1} \frac{\alpha_j}{(\alpha_j + \cdots + \alpha_M)} \mathrm{e}^{-(\alpha_l + \cdots + \alpha_M)s}\, ds \\
&\quad = \prod_{j=l}^{M-1} \frac{\alpha_j}{(\alpha_j + \cdots + \alpha_M)} \alpha_{l-1} \int_t^\infty \mathrm{e}^{-(\alpha_{l-1} + \alpha_l + \cdots + \alpha_M)s}\, ds \\
&\quad = \prod_{j=l-1}^{M-1} \frac{\alpha_j}{(\alpha_j + \cdots + \alpha_M)} \mathrm{e}^{-(\alpha_{l-1} + \alpha_l + \cdots + \alpha_M)t}.
\end{aligned}$$

Now the assertion follows from (29) with $l := 1, t := 0$ □

References

[Dev86] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York, 1986.

[Knu69] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., 1969.

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[Li94] Kim-Hung Li. Reservoir sampling algorithms of time complexity $O(n(1 + \log(N/n)))$. *ACM Transactions on Mathematical Software*, 20(4):481–493, December 1994.

[Ree03] Colin Reeves. Genetic algorithms. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2003.

[Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985.