Universität Karlsruhe (TH)
Institut für Theoretische Informatik

Min-Wise Independent Permutation Families
Analysis and Optimization

Studienarbeit von Tobias Matzner (Mat. Nr. 1053921)
Betreuer Prof. Dr. D. Wagner

Hiermit erkläre ich, die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen verwendet zu haben.

Karlsruhe, den 9. November 2005 ....

Abstract

This work introduces the concept of min-wise independent permutation families and their use for various applications to handle large data. We analyze the theory and implementation of an existing library and document its optimization and testing. Furthermore we give some hints to recent developments in the field of minwise hashing and related concepts that eventually could lead to faster implementations.

Contents

1 Introduction **4**

2 Minwise independent permutation families **5**

3 Sample Applications **6**

3.1 Estimating the similarity of web pages 6

3.2 Estimating rarity of elements in data streams 7

4 Optimization of a min-wise independent permutation library **8**

4.1 Analysis of the implementation 8

4.2 The landmark storage 10

4.3 Complexity analysis 11

4.4 Removing modulus calculations 13

4.5 Results of the optimization 15

5 Application in a streaming algorithm and tests **15**

6 Research **19**

6.1 A family based on geometric rectangles 19

6.2 Properties of min-wise independent permutations 20

7 Conclusions **21**

1 Introduction

In the analysis and application of hashing functions a property often resorted to is that, given the sets A and B, the hashing function considered is chosen at random from all possible functions $A \to B$. For a “classical” example see the work of Knuth on Sorting and Searching [8]. However the specification of such a function needs $|A| log |B|$ bits[1], which is too much for most practical situations, where hashing is usually used to handle the big set A efficiently. This led to the concept of *universal hashing.* Although this notion is used in slightly different meanings,[2] it generally implies the concept of choosing from a smaller set of hashing functions that behave under certain conditions given in most situations as if they were selected randomly from all possible functions. For example in [2] *universal hashing functions* are defined as functions for which holds $Pr(h(x_1) = y_1 \text{ and } h(x_2) = y_2) \leq \frac{1}{|B|^2}$ where $h \in A \to B$, for any x_1 and x_2, y_1 and y_2 arbitrary. There exist functions that have this property – which would also result from truly random selection – and that have only size $O(|B|^2)$. Oestlin and Pagh in [10] included a list of pertinent publications concerning such families.

Following these results Broder, Charikar, Frietze and Mitzenmacher in [2] define *min-wise independent permutations.* They are useful for example in streaming algorithms, (e.g. estimating the rarity of elements in a data stream) or elsewhere when a huge amount of data is to be dealt with (e.g. estimating the similarity of web-pages by search engines). We will present these two examples more circumstantially below. Their practicability stems from the fact that in respect to certain qualities, which often are the only ones really required by realistic applications, they also behave like truly random functions. This enables their use in approximation schemes exploiting the fact that noticeable particularities of their results allow to infer certain properties of their input data: Random functions with random input should have a more or less equally distributed result. If now certain results are more frequent than others and the function behaves as if it was random it can only mean that the input has certain characteristics that lead to the observed behavior.

In section 2 we give exact definitions and preliminaries for those functions. Section 3 presents some sample applications of minwise independent permutations. In section 4 we present the analysis and optimization of a li-

[1]cf. [2] p. 2

[2]Carter and Wegman who introduced this term in [5] applied it to functions with certain defined properties whereas for example according to the NIST *Dictionary of Algorithms and Data Structures* [7] it only denotes the random choice from an arbitrary set of hash functions.

brary implementing such a family whose testing and application is described in section 5. Finally we present some recent results in section 6 that could lead to faster implementations.

2 Minwise independent permutation families

We intend with $[n]$ the set of natural numbers $\{1...n\}$ and with S_n the set of all possible permutations on this set. A family of permutations $\mathrm{F} \subset S_n$ is defined to be *min-wise independent* if for any set $X \subseteq [n]$ and any $x \in X$, when $\pi \in \mathrm{F}$ is chosen at random we have

$$Pr(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|} \tag{1}$$

or in other words any element in X has an equal chance of being mapped to the minimum by the permutation chosen.

For practical purposes it is not necessary that this property holds exactly. *Approximately min-wise independent permutations* are used instead. These are families $\mathrm{F} \subset S_n$ for which holds for any set $X \subseteq [n]$ and any $x \in X$, when $\pi \in \mathrm{F}$ is chosen at random

$$\left| Pr(min\{\pi(X)\} = \pi(x)) - \frac{1}{|X|} \right| \leq \frac{\epsilon}{|X|} \tag{2}$$

where ϵ is called the *relative error.* We are also seldom interested in permutations of the entire universe (i.e. the set $[n]$) or large portions of it. Instead, we want to calculate the position of only a few values after the permutation. Thus, we define a family $\mathrm{F} \subset S_n$ of permutations to be *min-wise independent for sets up to size k* if for any set $X \subseteq [n]$ with $|X| \leq k$ and any $x \in X$, when $\pi \in \mathrm{F}$ is chosen at random

$$Pr(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}, \quad |X| \leq k \tag{3}$$

The exact min-wise property (1) needs exponential size, whereas the approximate property (2) can be satisfied by polynomial size families,[3] that hence can be implemented more efficiently. Also the restriction of the set size allows saving of computation time[4].

[3]see [2] for more specific results

[4]Note the factor k in the complexity data in section 4

3 Sample Applications

To give a better intuition of the benefits of the aforementioned properties we present two applications of min-wise independent permutations. We describe the techniques always in terms of exact independence to simplify the argument, an implementation however would use permutations with both relaxations.

3.1 Estimating the similarity of web pages

The search engine *Altavista* applied min-wise independent permutations to get an efficient estimate of the similarity of the pages in its cache. Although we do not know of any current application of this concept its simplicity makes it an alternative being worth to be considered to handle huge amounts of documents. This was the original application that led to the idea of min-wise independent permutations as described in [2] whereon we base the following presentation. Similar ideas however have been conceived independently in several works, see [4] p.20 for an overview.

The relevant property of min-wise independent permutations is: given two sets A and B, π chosen at random from a min-wise independent family

$$Pr(min\{\pi(A)\} = min\{\pi(B)\}) = \frac{|A \cap B|}{|A \cup B|} \tag{4}$$

From (1) we have that for a fixed element the probability of becoming minimum is the inverse of the set size which is now $\frac{1}{|A \cup B|}$. As $min\{\pi(A)\} = min\{\pi(B)\}$ if and only if the minimum is in $A \cap B$ we have $|A \cap B|$ possible elements and (4) follows.

It is relatively easy to transform web pages into subsets of $[n]$ by associating each word with a certain number. The ratio $\frac{|A \cap B|}{|A \cup B|}$ can then be used as a sensible measure of their similarity. There exist more refined methods for constructing subsets of $[n]$ from documents for the use with this measure, for an example see [3]. Now it suffices to compute for each document the set $D \subset [n]$ and then calculate and store the list $(min\{\pi_1(D)\}...min\{\pi_l(D)\})$. The fraction of common elements in the corresponding lists of two web pages can then be used as an approximation of their similarity. The number l of permutations used depends on the desired accuracy of the estimate. The authors of [2] suggest using 100 permutations. So the comparison of only 100 integer values is enough to give an estimate of the similarity, which is a very small effort compared to dealing with the documents.

3.2 Estimating rarity of elements in data streams

M. Datar and S. Muthukrishnan present an algorithm to estimate the rarity of elements in a data stream whereof we want to give a basic sketch. For details see [6].

Very generally speaking, streaming algorithms get as input a stream of data items whose summed size exceeds the available memory and thus have to be processed "on the fly" with limited time and space per item. For a good introduction on these algorithms see [9]. As it is our intention to show the efficiency of computations using min-wise independent permutations we will present the algorithm on a level of abstraction such that a thorough knowledge of streaming algorithms is not necessary. This algorithm is also presented because enabling its implementation was the primary purpose of our optimization of the permutation library described below.

Let S be a multiset containing all the elements of the stream. Let D be the set of all distinct elements in S and $R_\alpha = \{i | i \in D$, multiplicity of i in S is $\alpha\}$. Then the *α-rarity* is the ratio $\rho_\alpha := \frac{|R_\alpha|}{|D|}$ which we want to compute.

Since $R_\alpha \subset D$ holds: $\frac{|R_\alpha \cap D|}{|R_\alpha \cup D|} = \frac{|R_\alpha|}{|D|} = \rho_\alpha$ and thus, using again (4), we have $Pr(\ min\{\pi(R_\alpha)\} = min\{\pi(D)\}\ ) = \rho_\alpha$. So again simply maintaining hash values suffices for an estimation. Moreover, $\pi(R_\alpha) = \pi(D)$ if and only if the element of D that became minimum after permutating is contained in R_α. Accordingly, we need only to calculate the values for D. This yields the following algorithm:

Given an input stream of data items $a_1, a_2, ...,$

- Chose k permutations $\pi_1 ... \pi_k$
- Per permutation we maintain a counter $c_i(t)$ and a variable $\pi_i^* := min_{a_j, j<t}\{\pi_i(a_j)\}$ that stores the current minimum result of the j-th permutation.
- For each new data item a_t the permutations are calculated. If a value $\pi_i(a_t)$ is strictly smaller than the current minimum $\pi_i^*(t-1)$ it is updated and the corresponding counter $c_i(t)$ is set to zero
- When the current minimum is observed again (i.e. $\pi_i(a_t) = \pi_i^*(t-1)$) the corresponding counter is incremented

Now the fraction of counters that have value α,
$\hat{\rho}_\alpha(t) := |\{l | 1 \leq l \leq k, c_l(t) = \alpha\}|/k$ is an approximation of the α-rarity ρ_α. The number k of permutations used depends on the desired accuracy of the estimate.

4 Optimization of a min-wise independent permutation library

At the time of writing, the only freely available implementation of a min-wise independent permutation family we know of is by Jerry Zhao[5], which offers a C++ library as well as some code to test the statistical properties. It has a good space complexity of $O(loglog\ n + log\ k + log\frac{1}{\epsilon})$ and a time complexity of $O(n(loglog\ n + log\ k + log\frac{1}{\epsilon}))$ for calculating the position of one value after the permutation on $[1...n]$ with independence restricted to sets of size k and error ϵ.[6] However, it is too slow for practical purposes; its optimization will be described in the following section.

4.1 Analysis of the implementation

The permutations are implemented using a combination of several *k-wise independent hashing functions*. Such a hashing function behaves like a truly random function on sets up to size k. They also are implemented with a certain error ϵ permitted, in analogy to the permutations. Each of those functions decides a certain number of bits of the permutation value, where this number is reduced recursively, allowing the parameter k – and thus space and time cost – to be less for each recursion.

For constructing a permutation on $[n]$, restricted to k_{perm} with error ϵ, we use a certain number of hash functions h_i that map from $[n]$ to a set $[r_i]$. The function h_1 is restricted to k_{perm}, function h_{i+1} to $h_i/2$. The sets $[r_i]$ are chosen so that $log\ r_i \geq 2 + \frac{2}{k_i}\left(log\frac{2}{\epsilon} + log\ log\ k_{perm}\right)$. Broder, Charikar, Frieze and Mitzenmacher show[7] that this yields the desired permutation, taking the result of h_1 as the most significant bits of the permutated value, that of h_2 as the second most significant bits and so on recursively until all required bits are calculated. That way also the required number of hashing functions is determined.

This construction is implemented in the class `mwPermutation`, whereas the k-wise independent hashing functions are implemented in the class `kwHashFunction`. The right hand side (classes drawn black) of figure 2 shows the composition of the library in UML. Profiling indicates that most of the computation time is used for calculating the single k-wise independent hashes, whereas combining them does not offer much possibility for optimization. So the focus of the remainder of this section will lie on these

[5]`http://www.icsi.berkeley.edu/~zhao/minwise/`

[6]ibid.

[7][2] p.12 et seqq.

computations.

The k-wise independent hashes are constructed following [1] using a *linear feedback shift register(LFSR)*:

Given two bit-sequences, the random *start sequence* $s = s_0, ..., s_{m-1}$ and the *feedback rule* $f = f_0, ..., f_{m-1}$, where $f(t) := t^m + \sum_{j=0}^{m-1} f_j \cdot t^j$ is an irreducible polynomial, they generate a *shift register sequence* $r = r_0, ...r_p$:
$r_i = s_i$ for $i < m$
$r_i = \sum_{j=0}^{m-1} f_j \cdot r_{i-m+j}$ for $i > m$

The n-bit hash-value of a number x is calculated by interpreting the n bits from $r_{n \cdot x}$ onwards as digits of a binary number. The length m depends on the desired size and accuracy of the hashing function: For a shift register sequence of p bits which are ϵ-away (in L_1 norm) from k-wise independence $m = 2(\lceil \frac{k}{2} + log\frac{1}{\epsilon} + log(1 + \frac{(k-1)t}{2})\rceil)$ is necessary.

This is implemented in the class `LFSR`. Profiling reveals that the calculation of this sequence takes up the major part of computation time. Figure 1 shows the output of the *gprof* profiler[8] for the permutation of 4000 values from a test data set, that would also be used to test streaming algorithms as the one described above. The permutations were on $[0..2^{32} - 1]$ restricted to pairs ($k = 2$) with error $\epsilon = \frac{1}{32}$, choosing also here a configuration common the application in the algorithm of section 3.2. `__moddi3` is a *libc*-function, which implements modulus calculations for `long long int` variables, so we didn't see much room for improvement there. However, we succeeded in omitting a huge part of these costly calculations, as described below. Nearly all of the calls of `__moddi3` are in `LFSR::getBits`, which thus takes up virtually all of the computation time, so the optimization has to focus on this function.

The function `LFSR::getBits` implements the calculation of the LFSR sequence, using an array of length $m + 1$, which is sufficient as the preceding m bits are enough to calculate the next bit in the sequence, which can be stored in the $(m + 1)$th position. A pointer $modulo\ m$ circles on the array thus simulating an infinite sequence. This is a very memory efficient design, especially considering that the shift register sequence can theoretically grow up to a length of 2^{64} bits, which cannot be stored. However, the calculation has to restart every time the function is called from the first m bits, i.e. the start sequence s. So the first optimization step is to include a storage for

[8]The names of the functions where edited for better readability. All tests concerning computation times where performed on a Celeron 700 MHz with 192 MB RAM, linux operating system and GCC 3.3.6

intermediate results.

```
Flaches Profil:

Jedes Muster zählt als 0.01 seconds.
  %    kumulativ   Selbst            Selbst   Gesamt
 Zeit   seconds   seconds  Aufrufe  s/Aufru  s/Aufru  Name
 72.44    608.51   608.51                             __moddi3
 27.54    839.82   231.31    39990     0.01     0.01  LFSR::getBits
  0.01    839.92     0.10    39990     0.00     0.01  mwPermutation::map
  0.00    839.95     0.03        1     0.03   231.45  testfile
 [...]
  0.00    840.01     0.01      167     0.00     0.00  irreduciblePolynomial::iptest
 [...]
  0.00    840.02     0.00    39990     0.00     0.01  kwHashFunction::hash
  0.00    840.02     0.00     7267     0.00     0.00  RNGImplementation::next
  0.00    840.02     0.00       10     0.00     0.00  mwPermutation::mwPermutation
  0.00    840.02     0.00       10     0.00     0.00  kwHashFunction::kwHashFunction
  0.00    840.02     0.00       10     0.00     0.00  irreduciblePolynomial::getBits
  0.00    840.02     0.00       10     0.00     0.00  irreduciblePolynomial::irreduciblePolynomial
  0.00    840.02     0.00       10     0.00     0.00  LFSR::LFSR
  0.00    840.02     0.00        1     0.00     0.00  RNG::set_seed
  0.00    840.02     0.00        1     0.00   231.45  main
```

Figure 1: Profiling results of the original implementation

4.2 The landmark storage

We decided against a results cache in favor of a design more independent of the locality of potential application data. The basic idea is to periodically store blocks of m bits, which we call *landmarks*. These allow to resume the calculation at the respective position in the shift register sequence, omitting the need of restarting from the very beginning.

The landmark storage is implemented as a separate class `lmStorage`, which has at its center a `bitvector`-object that wraps a `vector` from the *Standard Template Library*, conveniently allowing to store and retrieve single bits. The class `LFSR` is extended by a `lmStorage` member object that offers five public functions:

- `checkRecord()` checks if the current position in the shift register sequence is part of a landmark block and thus has to be stored. It also checks if enough bits will be calculated to fill the landmark entirely.
- `recordValue()` saves the calculated value in the landmark storage.

These two functions are included in the loop calculating the LFSR values. So every time a request is made for bits with an index higher than the last landmark block, the storage is automatically extended.

- `getMax()` returns the position in the shift register that corresponds to the last landmark block. If calculations are below this point, `checkRecord()`is disabled as no new landmark blocks will be passed.

- `getPos()` Given the position of the requested bits, this function returns the position of the next landmark below where calculation can be resumed to arrive at the desired bits.

- `load()` loads the landmark indicated by `getPos()` into the array and sets all pointers and index variables to the correct values to resume calculation.

These functions are called at the very beginning of the function calculating the LFSR sequence, assuring that only the minimum of computation needed is performed.

The interface of the `LFSR` class remains unchanged, as well as the other classes of the min-wise library. Figure 2 shows the integration of the new classes (drawn in blue) in UML. The only change for the user is the need to specify an additional constant `LANDMARK_DISTANCE`, which specifies in blocks of size m the distance between the landmarks; i.e. a value of 1000 means that every 1000th block of m bits in the LFSR sequence is stored. Obviously, the gain in computation time increases with a smaller distance, but memory usage increases. A "good" value for this parameter depends on two parameters: the amount of memory which the user wants to dedicate to the program and the highest position of requested bits in the sequence, which is in the paradigm of streaming algorithms unknown to the program as a matter of principle. Thus we decided to leave the decision to the user. An alternative would be to only ask for (or even estimate) the amount of available memory and spread the landmarks evenly, assuming that the entire sequence will be needed.

4.3 Complexity analysis

The running time of the two function calls included in the sequence calculation, `checkRecord()` and `recordValue()`, is independent of input values. The reading of m bits at the beginning to resume the calculation is the same effort as reading the start sequence, plus a constant time to compute the position of the landmark at which to resume. Thus the time complexity of the calculation remains the same.

The space required is the amount needed by the original implementation plus m times the number of landmark blocks stored, which is limited by the

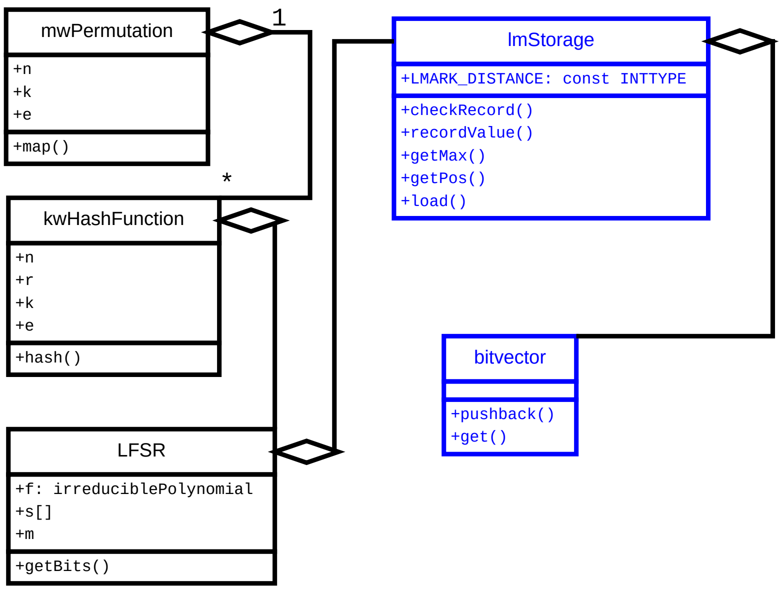

Figure 2: UML-Diagram of the optimized library. Existing classes drawn in black, added classes in blue.

```
Flaches Profil:

Jedes Muster zählt als 0.01 seconds.
  %    kumulativ   Selbst              Selbst    Gesamt
 Zeit   seconds   seconds   Aufrufe  s/Aufru   s/Aufru   Name
 72.27     38.37    38.37                                  __moddi3
 26.86     52.63    14.26     39990     0.00      0.00   LFSR::getBits
  0.36     52.82     0.19     39440     0.00      0.00   LmStorage::load
  0.32     52.99     0.17                                  __divdi3
  0.08     53.03     0.04     39990     0.00      0.00   mwPermutation::map
 [...]
  0.02     53.06     0.01     39880     0.00      0.00   LmStorage::getPos
  0.02     53.07     0.01       167     0.00      0.00   irreduciblePolynomial::iptest
 [...]
  0.00     53.09     0.00    403870     0.00      0.00   LmStorage::recordValue
  0.00     53.09     0.00    403100     0.00      0.00   LmStorage::checkRecord
  0.00     53.09     0.00     40100     0.00      0.00   LmStorage::getMax
  0.00     53.09     0.00     39990     0.00      0.00   kwHashFunction::hash
  0.00     53.09     0.00      7267     0.00      0.00   RNGImplementation::next
 [...]
  0.00     53.09     0.00        10     0.00      0.00   mwPermutation::mwPermutation
  0.00     53.09     0.00        10     0.00      0.00   kwHashFunction::kwHashFunction
  0.00     53.09     0.00        10     0.00      0.00   irreduciblePolynomial::getBits
  0.00     53.09     0.00        10     0.00      0.00   irreduciblePolynomial::irreduciblePolynomial
  0.00     53.09     0.00        10     0.00      0.00   LFSR::LFSR
  0.00     53.09     0.00        10     0.00      0.00   LmStorage::LmStorage
  0.00     53.09     0.00         1     0.00     14.51   testfile
  0.00     53.09     0.00         1     0.00      0.00   RNG::set_seed
  0.00     53.09     0.00         1     0.00     14.51   main
```

Figure 3: Profiling results using the lmStorage

memory and the constant `LANDMARK_DISTANCE`. Accordingly also the space complexity remains in the same magnitude.

4.4 Removing modulus calculations

Figure 3 shows the profiling output after inclusion of the lmStorage-Object when called with the same parameters as used for figure 1.

It is possible to observe the fast performance of the `lmStorage` functions and the reduced amount of LFSR computation time (here quite significantly using a landmark distance of only ten, which usually is too memory consuming for practical purposes). Yet, the corresponding function still consumes most of the time due to its calling of the costly `__moddi3` function for the purpose of computing index positions in the array storing the LFSR sequence: The position in the array is the position in the sequence modulus m. We replaced these calls by `if-then` statements. To demonstrate the efficiency of this technique we used the toy programs shown in figures 4 and 5 that perform the same computation.

```
int k=0;
int a=0;

for(long long int i=0; i<NUM_IT; i++){
    a+=(i%2);
}
```

Figure 4: Loop using modulus

```
long long int k=0;
int a=0;

for(int i=0; i<NUM_IT; i++){
    k++;
    if(k==2){k=0;}
    a+=k;
}
```

Figure 5: Loop using if-then statements

Even though they show the worst case using modulus 2 which means that in figure 5 the variable `k` has to be reset the largest number of times possible (and the condition `k==0` changes very often between true and false irritating branch prediction mechanisms) the loop in figure 5 took 25.240 seconds to compute compared to 44.650 seconds for the loop in figure 4 ($1.024 \cdot 10^6$ iterations). Thus all of the modulus calculations where replaced with *if-then* statements, apart from initially computing the start values from the function parameters. Those where stored in additional variables reducing further the amount of `__moddi3` calls.

```
Flaches Profil:

Jedes Muster zählt als 0.01 seconds.
  %    kumulativ   Selbst             Selbst   Gesamt
 Zeit   seconds   seconds  Aufrufe  s/Aufru  s/Aufru  Name
 93.24      4.14      4.14    39990     0.00     0.00  LFSR::getBits
  2.25      4.24      0.10                             __moddi3
  1.80      4.32      0.08    39440     0.00     0.00  LmStorage::load
  0.90      4.36      0.04    39990     0.00     0.00  mwPermutation::map
  0.68      4.39      0.03   403870     0.00     0.00  LmStorage::recordValue
  0.45      4.41      0.02      167     0.00     0.00  irreduciblePolynomial::iptest
  0.23      4.42      0.01   403100     0.00     0.00  LmStorage::checkRecord
  0.23      4.43      0.01    39880     0.00     0.00  LmStorage::getPos
 [...]
  0.00      4.44      0.00    40100     0.00     0.00  LmStorage::getMax
  0.00      4.44      0.00    39990     0.00     0.00  kwHashFunction::hash
  0.00      4.44      0.00     7267     0.00     0.00  RNGImplementation::next
 [...]
  0.00      4.44      0.00       10     0.00     0.00  mwPermutation::mwPermutation
  0.00      4.44      0.00       10     0.00     0.00  kwHashFunction::kwHashFunction
  0.00      4.44      0.00       10     0.00     0.00  irreduciblePolynomial::getBits
  0.00      4.44      0.00       10     0.00     0.00  irreduciblePolynomial::irreduciblePolynomial
  0.00      4.44      0.00       10     0.00     0.00  LFSR::LFSR
  0.00      4.44      0.00       10     0.00     0.00  LmStorage::LmStorage
  0.00      4.44      0.00        1     0.00     4.33  testfile
  0.00      4.44      0.00        1     0.00     0.00  RNG::set_seed
  0.00      4.44      0.00        1     0.00     4.33  main
```

Figure 6: Profiling results with removed modulus calculations

Figure 6 shows the now increased percentage of time used for `LFSR::getBits`, but the huge gain in overall computation speed.

We also implemented another version using two alternative LFSR::getBits functions, one using `long long int`, the other `long int` variables and a function pointer switching between them, so that the cost intensive variables would only be used when needed. In the `long long int` version the modulo calls where substituted as mentioned above. This however gained very little time. (Seemingly the increased object complexity cancels the gain.) We decided not to pursue this approach esteeming clarity of design more than a few milliseconds spared.

4.5 Results of the optimization

We summarize the profiling results shown in figures 1, 3 and 6 and add results of a second run using the higher landmark distance of 100 blocks.

version	runtime	speedup	
original	840.02 s		overall speedup: 189,21
with landmarks	53.09 s	15.82	
with reduced mods	4.44 s	11,96	

Table 1: Results with `LANDMARK_DISTANCE 10`

version	runtime	speedup	
original	840,02 s		overall speedup : 20.02
with landmarks	532.81 s	1.58	
with reduced mods	42.09 s	12,67	

Table 2: Results with `LANDMARK_DISTANCE 100`

Note that the computation time scales quasi linear with the number of landmarks stored, which shows both the functioning and the space-time tradeoff of our optimization.

5 Application in a streaming algorithm and tests

Now with a minwise library in hand running in reasonable time, together with Dr. Buriol of the *Algorithm engineering group* at the *Department of Computer and System Sciences, University of Rome "La Sapienza"* we experimented with using the min-wise library in an implementation of the rarity algorithm introduced in section 3.2. However, the counters storing the number of times the minimum was observed increased much more often than this should happen, considering the test data. This led us to testing the performance of the library in respect to the statistical requirements it was supposed to meet.

For the tests we generated a set X of random values which were permutated by p min-wise independent permutations, restricted to sets of size at least $|X|$. For each value we recorded the number of times m it became minimum after the permutation. From (2) follows

$$\epsilon \leq \left| \left(Pr(min\{\pi(X)\} = \pi(x)) - \frac{1}{|X|} \right) \cdot |X| \right|$$

thus we calculated

$$\epsilon = \left| \left(\frac{m}{p} - \frac{1}{|X|} \right) \cdot |X| \right| = \left| \left(\frac{m \cdot |X|}{p} - 1 \right) \right|$$

We present the results from a test run on sets of eight values and requested error below $\frac{1}{32}$ using 6000 permutations on [0...255] (For observing statistical properties we had to use this relatively small permutation space to cover a sufficiently large fraction of all possible permutations.)

In figure 7 the data for one value is shown, in figure 8 the mean error of all values, plotting the results after each 100 permutations performed. The error stays below the required value after circa 2500 permutations and stabilizes at a level well below the requirement after around 5000 permutations. The latter result was confirmed by a second test run on 20000 permutations whose results are shown in figure 9.

A second test run was done using the same parameters as before but permutations on $[0..2^{16} - 1]$ instead of [0..255]. As figure 10 shows, having now a bigger sample space, we needed more permutation runs until we could observe the error value stabilizing. However, also in this case it eventually stays way below the expected value. We conclude that the library fulfills the statistical properties required well.

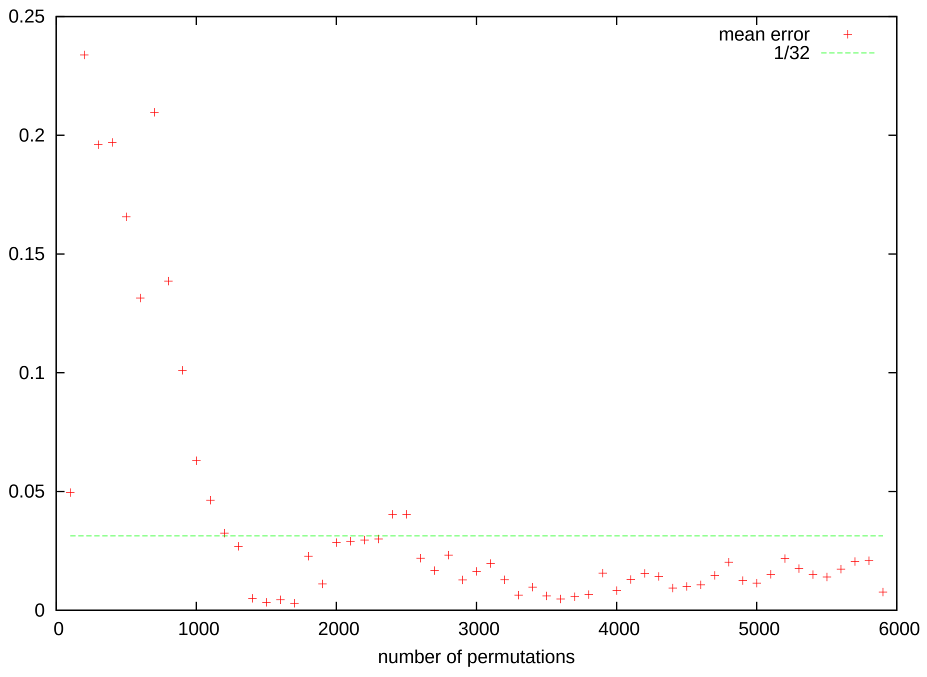

Figure 7: Error of one test value

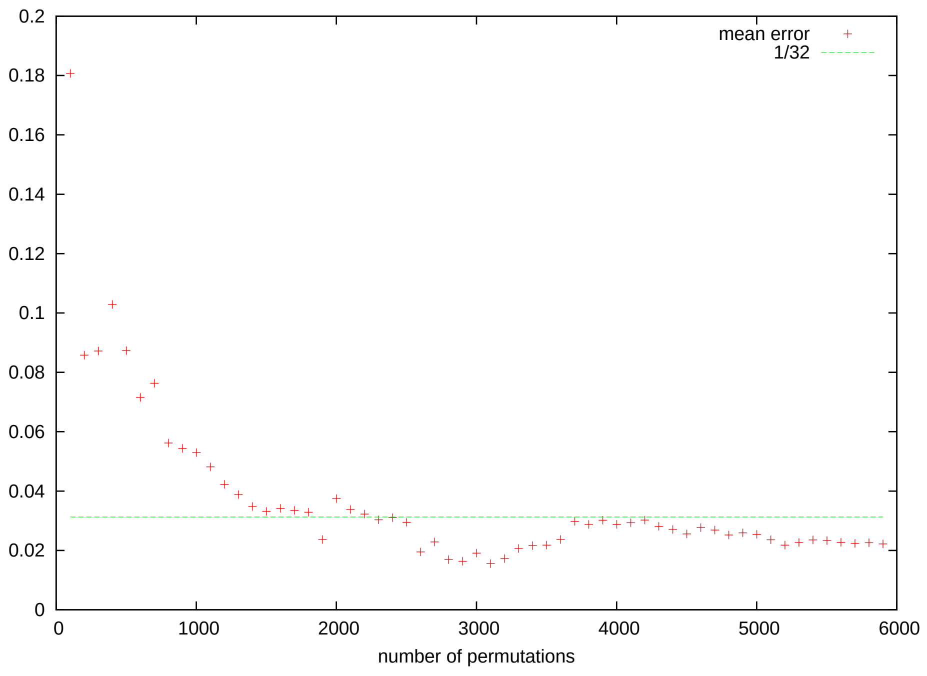

Figure 8: Mean error of all test values

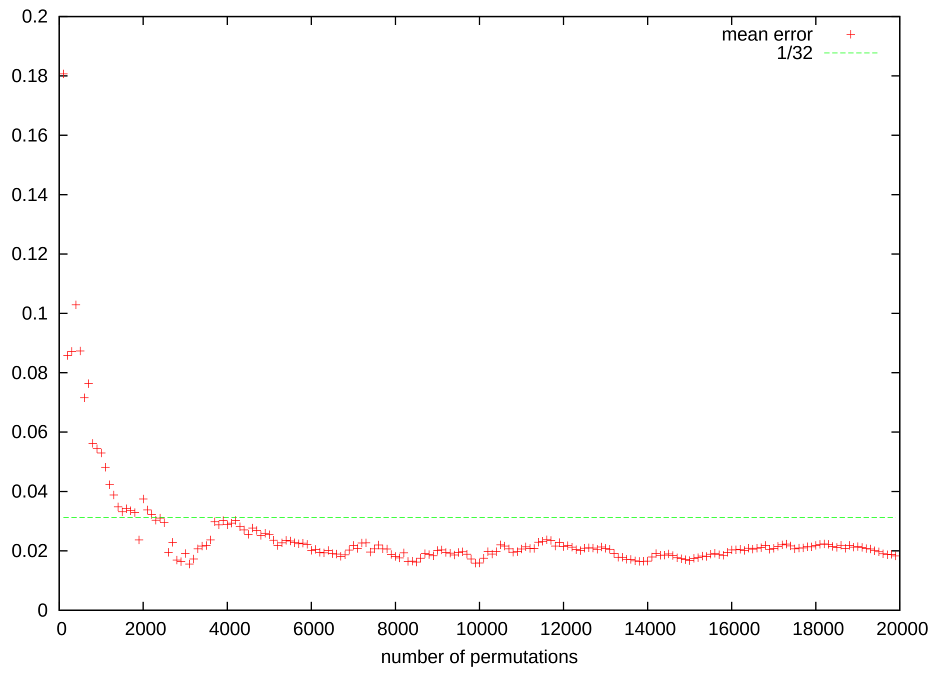

Figure 9: Mean error of all test values on more permutations

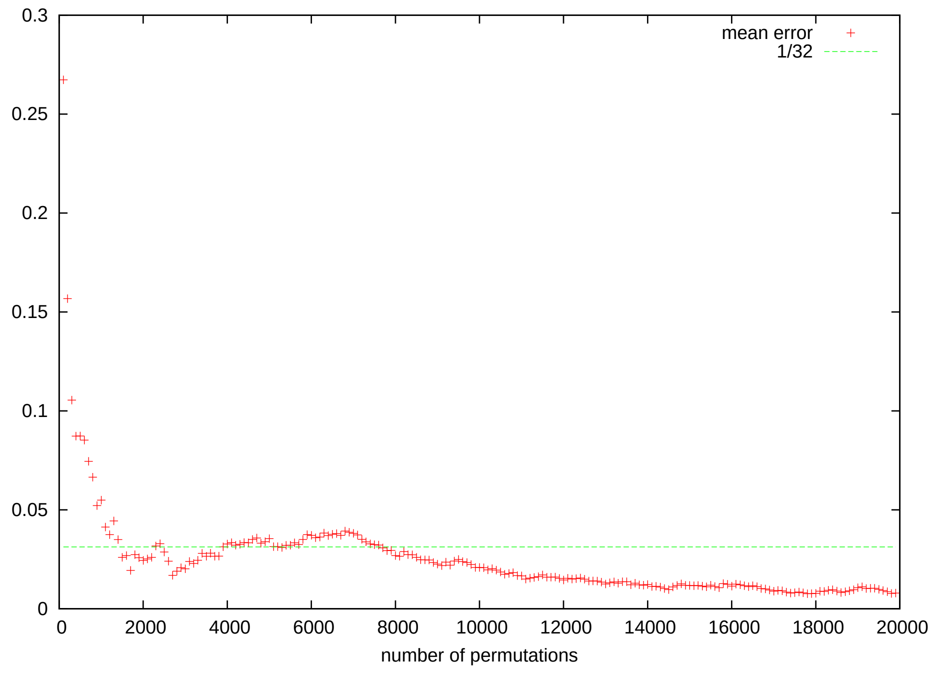

Figure 10: Mean error of all test values (permutations on $[0..2^{16}-1]$)

6 Research

Even considering the improvements achieved optimizing the library, it still runs several hours for applications as the rarity algorithm presented above. Jerry Zhao suggests a re-implementation using a fast mathematics library on his homepage. There also exist several interesting recent results concerning min-wise independent permutations, both theoretically and constructive. We could not further investigate into implementing another version or permutation family in the scope of this work. However, we would like to present some of these results that could lead to faster programs.

An important measure in the theoretical analysis of permutation families is their size, i.e. the cardinality of the set of permutations they define. The general goal is to find a family having the useful independence properties but being relatively small. As already mentioned in section 2 Broder, Charikar, Frietze and Mitzenmacher in [2] showed that an exact min-wise independent family[9] has at least a size of $e^{n-o(n)}$. In [12] it is shown that even $lcm(1,2,...,n) \geq e^{n-o(n)}$ is a lower bound[10]. The construction with relaxations used in the implementation we optimized (also from [2]) yields a family of maximal size $2^{4k+o(k)}k^{2\log(\frac{\log n}{\epsilon})}$.

6.1 A family based on geometric rectangles

In [11] Saks, Srinivasan, Zhou and Zuckerman use a different approach constructing a polynomial size min-wise independent family if $k \leq 2^{O(\log^{2/3} n)}$ and $\epsilon \geq 2^{-O(\log^{2/3} n)}$. Their family uses *geometric rectangles* defined as follows:

The set of geometric rectangles $\mathcal{GR}(m,d,n)$ is the set $[a_1,b_1) \times ... \times [a_n,b_n)$ where $i, a_i, b_i \in \{0,1,...,m-1\}$ with $a_i \leq b_i$ and $a_i = 0, b_i = m-1$ holds simultaneously for at least $n-d$ indices i. The *volume* of such a rectangle R is $vol(R) := (\prod_{i=1}^{n}(b_i - a_i))/m^n$.

Now let $D \subseteq [0,m)^n$ for which holds:

$$\forall R \in \mathcal{GR}(m,n,d) \quad \left| \frac{|D \cap R|}{|D|} - vol(R) \right| \leq \frac{1}{m^2} \tag{5}$$

Let $\pi_r \in S_n$ the permutation defined using $r = (r_1,...,r_n) \in D$ such that for any $0 \leq i,j \leq n-1, \pi_r(i) \leq \pi_r(j)$ if and only if $r_i \leq r_j$ or $r_i = r_j$ but $i \leq j$.

[9]In this section we use as before n, k, ϵ as in the definitions (1)-(3)

[10]cf. also [11] p. 2

Section 1 of [11] states that the multiset of all permutations defined using the elements in D and setting $m = 2d^2/\epsilon$ is a min-wise independent permutation family of size $L = n^{O(1)} \cdot (d/\epsilon)^{O\left(\sqrt{\log(max\{2,d/\log(1/\epsilon)\})}\right)}$. This is polynomial in n given the abovementioned restrictions for k and ϵ. Furthermore, a lot computational effort of this family, as for example assuring the discrepancy property (5), is initialization work. Given $log\ L$ random bits to index a permutation π of the family, it is possible to calculate $\pi(i)$ for a value $i \in [n]$ in time polylogarithmic in L .

6.2 Properties of min-wise independent permutations

In [4] Broder and Mitzenmacher analyze further the properties of min-wise independent permutations. They show that in fact any randomized sampling scheme used for testing relative intersection of sets based on equality of samples yields a min-wise independent family. More specifically, given a family $\mathcal{F}$ of functions mapping from nonempty subsets of $[n]$ to an arbitrary subset Ω, assume there exists a probability distribution on $\mathcal{F}$ such that when $f \in \mathcal{F}$ is chosen according to this distribution

$$Pr(f(A) = f(B)) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

for any sets A and B. Then there exists a min-wise independent permutation family $\mathcal{P}$ such that every $f \in \mathcal{F}$ is defined by

$$f(X) = f\left(\left\{\pi_f^{-1}(min\{\pi_f(X)\})\right\}\right)$$

for some $\pi_f \in \mathcal{P}$.

Accordingly, every method of sampling to estimate resemblance using 6 is equivalent to sampling with min-wise independent permutations. Note also, that the uniform distribution of the choice of the permutation, that we implied throughout this text, is not necessary for an estimation scheme to work. However, a uniformly distributed family obviously facilitates the work.

The choice of the minimum element as relevant is somehow arbitrary. In [4] it is proven that any min-wise independent permutation family is also *max-wise independent*. This led to a generalization of (1): With a fixed permutation $\sigma \in S_n$, π chosen at random, it is required:

$$Pr\left(min\{\sigma(\pi(X))\} = \sigma(\pi(x))\right) = \frac{1}{|X|} \tag{7}$$

Max-wise independence for example can then be obtained choosing $\sigma(i) = n + 1 - i$. The choice of another order σ instead of the minimum does not improve anything. However there exist families that have property (7) with respect to several orders. For some n exist even families with (7) respect to all possible orders on $[n]$. This yields the possibility of extracting a sample for several orders avoiding costly computations of $\pi(X)$. A point that needs attention however, is the need to find a bound for the correlation these samples will obviously have.[11]

7 Conclusions

We gave an introduction to the concept of minwise independent permutations and their properties. They make them useful to create highly efficient estimates to extract information from large data sets. Two of these applications where presented: estimating the similarity of web pages and estimating the rarity of elements in a data stream. We showed in detail how a special permutation family of this kind is constructed from k-wise independent hashing functions, that in turn are built on a linear feedback shift register. Also its implementation was analyzed, which gave way to a quite successful optimization. The correct functioning of the implementation was established in several tests. However, the resulting library is slow used in the desired applications. We presented some recent research results, that could lead to an improvement, where especially the permutation family based on geometric rectangles seems promising. It has linear time complexity admitting an error not too large and it should be possible to implement rather efficiently.

[11]cf. [4] p. 20 et seq.

References

[1] N. Alon, O. Goldreich, J. Håstad, and R. Peralta. Simple constructions of almost k-wise independent random variables. *Journal of Random structures and Algorithms*, 3(3):289–304, 1992.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*, 1998.

[3] A. Z. Broder, S. C. Glassmann, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, 1997.

[4] A. Z. Broder and M. Mitzenmacher. Completeness and robustness properties of min-wise independent permutations. *Random Structures and Algorithms*, 18(1):18–30, 2001.

[5] J. L. Carter and M.N. Wegeman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[6] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *Proceedings of the 10'th European Symposium on Algorithms. Rome, Italy*, 2002.

[7] P. E. Black ed. Dictionary of algorithms and data structures. U.S. National Institute of Standards and Technology `http://www.nist.gov/dads`(2 November 2005).

[8] D. E. Knuth. *The art of computer programming, volume 3: sorting and searching.* Addison-Wesley, 2nd edition, 1998.

[9] S. Muthukrishnan. Data streams: algorithms and applications. `http://www.cs.rutgers.edu/~muthu/`, 2003.

[10] A. Ostlin and R. Pagh. Simulating uniform hashing in constant time and optimal space. In *Research report RS-02-27*. BRICS, Department of Computer Science, University of Aarhus, 2002.

[11] M. Saks, A. Srinivasan, S. Zhou, and D. Zuckerman. Low discrepancy sets yield approximate min-wise independent permutation families. *Information Processing Letters*, 73(1–2):29–32, 2000.

[12] Y. Takei, T.Itoh, and T. Shinozaki. An optimal construcion of exactly min-wise independent permutations. Technical Report COMP98-62, IE-ICE, 1998.