# Max-Minhash Clustering Algorithm of RNA sequences

Mehdi Asser Husum Nadif

April 24, 2015

# Contents

# Minhash

# Hashing

In the implementation of randomized algorithms, a strong need for data representation is needed. Truly independent hash functions are extremely useful in these situations. It allows generalization of arrays, where it is possible to access an arbitrary position in an array in *O(1)* time.

## Introduction to hash functions

Imagine a universe of keys $U = \{u_0, u_1, ..., u_{m-1}\}$ and a range $[r] = \{0, 1, ..., m-1\}$ where $u_i$ are integers module $m$. Given a hash function $h(x) \rightarrow [r]$ which takes any $u_i, i = 0, 1, ..m-1$ as argument, it should hold that $\forall u_i, \exists r_j \in [r]$ such that $h(u_i) = r_j$. What remains is to consider collisions, which we define as

$$\delta_h(x, y) = \begin{cases} 1 & \text{if } x \neq y \text{ and } f(x) = f(y) \\ 0 & \text{else} \end{cases} \tag{1}$$

for a given hashfunction $h$ and two keys $x$ and $y$. The goal of a hashing algorithm is then to minimize the number of collisions across all possible keys. A truly random hash function can assure that there are no collision at all. Unfortunately, to implement such a function would require at least $|U| \log_2 m$ bits[5], defeating the purpose of hash functions altogether. Fixed hashing algorithms have attempted to solve this problem. Unfortunately, its dependence on input causes a worst case average retrieval time of $\Theta(m)$.[2]

Universal hashing can circumvent the memory and computation cost of both random- and fixed hashing, without losing much precision. An introduction to universal hashing will follow, alongside two applications of said hashing which will be tested later in this paper.

## Universal hash functions

The first mention of universal hashing was in [1], in which they define universality of hash functions as follows:

Given a class of hash functions $H : U \rightarrow [r]$, $H$ is said to be universal if $\forall x \forall y \in U$

$$\delta_H(x, y) \leq |H|/|[r]|$$

where, with $S \subset U$

$$\delta_H(x, S) = \sum_{h \in H} \sum_{y \in S} \delta_h(x, y)$$

That is, $H$ is said to be universal if

$$\Pr_h[h(x) = h(y)] \leq 1/m \tag{2}$$

for a random $h \in H$. In many applications, $\Pr_h[h(x) = h(y)] \leq c/m$ for $c = O(1)$ is sufficiently low.

**Carter and Wegman[1]**

Given a prime $p \geq m$ and a hash function $h^C_{a,b} : [U] \to [r]$,

$$h^C_{a,b}(x) = ((a * x + b) \mod p) \mod m \tag{3}$$

where $a$ and $b$ are integers mod $m$, where $a \neq 0$. We want to prove that $h^C_{a,b}(x)$ satisfies Eq. 2; thus proving that it is universal.

Let $x$ and $y$ be two randomly selected keys in $U$ where $x \neq y$. For a given hash function $h^C_{a,b}$,

$$r = a \cdot x + b \mod p$$

$$q = a \cdot y + b \mod p$$

We see that $r \neq q$ since

$$r - s \equiv a(k - l)( \mod p)$$

must be non-zero since $p$ is prime and both $a$ and $(k - l)$ are non-zero module $p$, and therefore $a(k - l) > 0$ as two non-zero multiplied by each other cannot be positive, and therefore must also be non-zero module $p$. Therefore, $\forall a \forall b, h_{a,b}$ will map to distinct values for the given $x$ and $y$, at least at the mod $p$ level.

**Dietzfelbinger et al.[3]**

Also commonly refered to as **multiply-shift**, this state of the art scheme described in [3] reduces computation time by eliminating the need for the **mod** operator. This is especially useful when the key is larger than 32 bits, in which case Carter and Wegman's suggestion is quite costly[5].

Take a universe $U \geq 2^k$ which is all $k$-bit numbers. For $l = \{1, .., k\}$, the hash functions $h^D_a(x) : \{0, ..., 2^k - 1\} \to \{0, ..., 2^l - 1\}$ are then defined as

$$h^D_a(x) = (a \cdot x \mod 2^k) \div 2^{k-l} \tag{4}$$

for a random odd number $0 < a < 2^k$. $l$ is bitsize of the value the keys map to. The following C-like code shows just how easy the implementation of such an algorithm is

```
h(x)=(unsigned) (a*x) >> (k-l)
```

This scheme only nearly satisfies Eq. 2, as for two distinct $x, y \in U$ and any allowed $a$

$$\Pr_{h^D_a}[h^D_a(x) = h^D_a(y)] \leq \frac{1}{2^{l-1}} = \frac{2}{m} \tag{5}$$

If Eq. 5 is not sufficently precise, Wölfel [6, p.18-19] modified this scheme so that it met the requirement in Eq. 2. The hash function is then

$$h^D_{a,b} = ((a \cdot x + b) \mod 2^k) \div 2^{k-l}$$

where $a < 2^k$ is a positive odd number, and $0 \leq b < 2^{k-l}$. This way Eq. 2 is met for $x \neq y$ (mod $2^k$). For a proof of this, consult [6][1]. The C-like implementation shown below reveals that the modifications are only minimal

[1]The text is in german

```
h(x)=(unsigned)((a*x) + b) >> (k-l)
```

## Hash Performance Test

Two hashing functions are described in the Hashing section. In order two decide which of these I would use in the algorithm, a few tests were performed in order to determine the speed of both.

To perform these tests, I wrote a short java program. It randomized a given number of k-mer transformations, and then counted the number of nanoseconds it took the Carter Wegman- and multiply-shift hashing schemes to hash all the transformations. The multiply-shift algorithm had $k - l = 32$, so that the output hash values were 32 bit integers.

The results of the tests can be seen in Table , in whic

| k-mer size | 10 | | 20 | | 30 | |
|---|---|---|---|---|---|---|
| # of hasvalues | Mult.shift | Carter | Mult.shift | Carter | Mult.shift | Carter |
| 1 mio. | $10.3 \pm 0.2$ | $11.9 \pm 0.5$ | $10.3 \pm 0.1$ | $11.6 \pm 0.1$ | $10.5 \pm 0.1$ | $11.6 \pm 0.1$ |
| 10 mio. | $40.9 \pm 0.5$ | $42.1 \pm 0.2$ | $41.3 \pm 0.1$ | $42.1 \pm 0.4$ | $41.8 \pm 1.6$ | $42.4 \pm 0.5$ |
| 50 mio. | $175.5 \pm 3.4$ | $187.2 \pm 10.6$ | $177.5 \pm 2.1$ | $181.6 \pm 3.8$ | $176.2 \pm 1.3$ | $179.7 \pm 1.7$ |
| 100 mio. | $343.0 \pm 1.4$ | $377.5 \pm 14.0$ | $341.6 \pm 1.0$ | $364.1 \pm 12.6$ | $346.7 \pm 6.9$ | $355.8 \pm 3.5$ |

Figure 1: The number of *ms* it took multiply-shift and the Carter Wegman hashing schemes to run a given number of randomized k-mer transformations.

## References


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

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19 – 51, 1997.

[4] M. Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 496–497, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[5] M. Thorup. High speed hashing for integers and strings, 2014.

[6] P. Wölfel. *Über die Komplexität der Multiplikation in eingeschränkten Branchingprogrammmodellen*. PhD thesis, Dortmund, Univ, 2003.

# Appendices

## 1 - Plan for subjects

These plans are made according to the 8 subjects that are mentioned in the Midway Report. They were written as i had to commence each subject, so that i had a plan of actions to tackle the subjects.

1. (1) Test of universal hashing schemes
   **Date**: 22 April 2015.
   **Goal**: To test several hashing schemes, and see which is quickest.
   **Procedure**: I shall use the two hash functions i defined in my universal hashing section.

   For testing, i shall use three different fasta files of different sizes. These are to be found in the directory given by Sune, wherein a lot of fasta files are located.

   Before testing, i must program both hashing function. This should no be too difficult as i have a working prototype running, so all that has to be done is copy the working implementation of the first hash function and change is so that it applies to the other hash function.

   Finally i shall have my java program write into a file the results of each run, so that i can create a table of the runtimes of each hashfunction.

   When the table is complete i can begin commenting on my results and see which hashfunction to use.