Max-Minhash Clustering Algorithm of RNA sequences

Mehdi Asser Husum Nadif

May 19, 2015

Contents

Hashing **3**
Introduction to hash functions 3
Universal hash functions 3
Carter and Wegman[3] 4

Minwise Hashing **5**
Max-wise hashing 6
Combining Max-wise and Min-wise 6

Hash Performance Test **7**

Appendices **10**

Hashing

Hashing functions are an integral part of minwise hashing, as it uses the values of hashing functions to determine similarity between two sets. Therefore, it is necessary that we have a small insight into universal hash functions before moving on to minwise hashing in the next section.

Introduction to hash functions

Imagine a universe of keys $U = \{u_0, u_1, ..., u_{n-1}\}$ and a range $[r] = \{0, 1, ..., m-1\}$. Given a hash function $h(x) \rightarrow [r]$ which takes any $u_i, i = 0, 1, ..n-1$ 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[8], 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 $\Theta(n)$.[4]

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 [3], 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 $\forall x \forall y \in U$

$$\delta_H(x, y) \leq \frac{|H|}{|[r]|}$$

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

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

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[3]

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.[5]

Also commonly refered to as **multiply-shift**, this state of the art scheme described in [5] 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[8].

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 [10, 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 [10][1]. The C-like implementation shown below reveals that the modifications are only minimal

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

[1]The text is in german

Minwise Hashing

The core of the clustering algorithm in this paper will be based on the concepts of minwise hashing as described in [2]. Minwise hashing has repeatedly proven a powerful tool when comparing large sets of strings rapidly, especially for duplicate detection of long articles. The use of minwise hashing for rRNA sequences has already been done in [7], but not with the slight modification that will be described at the end of this section.

Introduction to Minwise Hashing

Let there be two sets A and B. To find the similarity between the two sets, minwise hashing uses is the Jaccard similarity measure, which is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{6}$$

To increase the speed of calculating the Jaccard similarity, it however uses hash functions to find the value. Let us observe how the hash functions are used in this case.

Min-wise Independency

Let $H : U \to [r]$ be a class of hashfunctions. Then for any set $X \subseteq [r]$ and any $x \in X$ and let $h \in H$ be chosen uniformly at random, it is considered minwise independent if

$$\Pr(h_{\min}(X) = h(x)) = \frac{1}{|X|} \tag{7}$$

where

$$h_{\min}(X) = \min\{\forall x \in X, h(x)\}$$

Meaning that all elements in X must have an equal probability of having the minimum value going through h. As seen in Eq. 2, this probability is reachable using universal hash functions, which is a great increase in speed over perfect hashing functions.

Min-wise sketch

For two sets A and B it has been proven in [1] that Eq. 7 can be linked to the Jaccard similarity in Eq. 6 as

$$\Pr(h_{\min}(A) = h_{\min}(B)) = \frac{|A \cap B|}{|A \cup B|} \tag{8}$$

For a random set S_1, we may create a table of random $h_{\min,i}, i = 1,..,k$ such that

$$\hat{S_1} = \{h_{\min,1}(S_1), h_{\min,2}(S_1), ..., h_{\min,k}(S_1)\}$$

We may then compute the similarity of two sets $\hat{S_1}$ and $\hat{S_2}$ defined by the above equation as

$$J(A,B) = \frac{1}{k} \cdot \sum_{i=1}^{k} (h_{\min,i}(S_1) = h_{\min,i}(S_2)) \tag{9}$$

where

$$(h_{\min,i}(S_1) = h_{\min,i}(S_1)) = \begin{cases} 1, & h_{\min,i}(S_1) = h_{\min,i}(S_2) \\ 0, & otherwise \end{cases}$$

which is called the **minwise sketch**. As we discussed earlier in the universal hashing section, there will be a slight error in the calculation of h_min. Therefore we must see what influence the size of k will have on the error. A proof of the error using Chernoff Bounds[2] is found in [9]. The result is that the relation between k and ϵ, the error, is

$$k = O\left(\log \frac{1}{\epsilon}\right)$$

Thus, k influences the error inversely exponentially, meaning that $k \approx 100$ should guarantee very small error.

Max-wise hashing

The aforementioned modification is one inspired by the method in the paper [6]. It is an extension of the minwise sketch where in addition to using the minwise independent sets, we add the maxwise independent set too. Very literally, this means that instead of using the minimum hashvalue, we use the maximal hashvalue such the maxwise independence means

$$\Pr(h_{\max}(X) = h(x)) = \frac{1}{|X|}, h_{\max} = \max\{\forall x \in X, h(x)\} \tag{10}$$

the Jaccard similarity measure for two sets A and B is

$$\Pr(h_{\max}(A) = h_{\max}(B)) = \frac{|A \cap B|}{|A \cup B|} \tag{11}$$

and finally for a random set S_1 we may create a table of random $h_{\max,i}, i = 1, ..., k$ such that

$$\tilde{S_1} = \{h_{\max,1}(S_1), h_{\max,2}(S_1), ..., h_{\max,k}(S_1)\}$$

This sketch alone is not very different from minwise, and first becomes interesting once combining the two sketches.

Combining Max-wise and Min-wise

There are two ways of combining the max-wise and the min-wise algorithm. One is the method in [6], where they halve the amount of hashfunctions, so that for $i = 1, .., k/2$

$$J(A,B) = \frac{1}{K}\sum_{i=1}^{K/2}(h_{\min,i}(A) = h_{\min,i}(B) + h_{\max,i}(A) = h_{\max,i}(B)) \tag{12}$$

[2] A probablistic method to find the exponentially decreasing bounds between two independent variates.

Let this method be called **Max-minwise halved sketch**. This method has been proven to be double as quick as the min-wise hashing, without loss of precision[6]. It is also shown in Lemma 2 in [6] that for $i = 1, .., k/2$

$$\Pr(h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B)) = \frac{|A \cap B| - 1}{|A \cup B| - 1}$$

Meaning that a collision between $h_{\min}$ and $h_{\max}$ is very unlikely.

Another method, which was developed in the course of this paper uses the following combination

$$J(A, B) = \frac{1}{k} \sum_{i=1}^{k} (h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B)) \qquad (13)$$

where

$$h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B) = \begin{cases} 1, & h_{\min,i}(S_1) = h_{\min,i}(S_2) \\ 1, & h_{\max,i}(S_1) = h_{\max,i}(S_2) \\ 0, & otherwise \end{cases}$$

Let this method be called **Max-minwise sketch**. This also has finds the Jaccard similarity by the following proof:

$$\begin{aligned} &\frac{1}{k} \sum_{i=1}^{k} (h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B)) = \\ &\frac{1}{k} \sum_{i=1}^{k} (J(A, B) | J(A, B)) = J(A, B) | J(A, B) = J(A, B) \end{aligned} \qquad (14)$$

the final two steps follow from Eq. 9 and Eq. 11. Therefore we see that this method also finds the jaccard similarity.

As one may have noted, the difference between **Max-minwise halved sketch** and **Max-minwise sketch** is that the first runs only half as many times as the second.

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[3], 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 was set to shift only if the input was longer than 32 bits. In this case, the shift was $2 \cdot k - 32$.

[3] which are 2·k bits long

The results of the tests can be seen in Table 1. As we can note, the Carter Wegman implementation is consequently smaller than multiply-shift, in addition to having larger margin of error. We may note that the difference in speed is most notable at $k = 10$, diminishing at higher k. It should be noted however that the differences in speed are less than 50 ms, even at 100 mio. input hash values. Also, as the time increases linearly with the number of input values, so will the difference in speed.

As the difference in speed is quite minimal, the method used in the algorithm will be the Carter Wegman scheme, as it is easier to implement for input of such different sizes as k-mers.

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

Table 1: *ms* for multiply-shift- and Carter Wegman hashing schemes to 4 different numbers of randomized k-mer transformations.

References

[1] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29, Jun 1997.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:327–336, 1998.

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

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

[5] 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.

[6] J. Ji, J. Li, S. Yan, Q. Tian, and B. Zhang. Min-max hash for jaccard similarity. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 301–309, Dec 2013.

[7] Z. Rasheed and H. Rangwala. A map-reduce framework for clustering metagenomes. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 549–558, May 2013.

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

[9] S. Vassilvitskii. Lesson 1: Duplication detection, 2011.

[10] 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 both 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.

   Result: Done on April 24. Found that the multiply-shift was a tad quicker. Had to use randomized k-mer transformations, for the reason given just below. **Bugs**: Found a problem of memory when the number of sequences was at 5 mio. The k-mer transformations took too much space. This is to be expected in a single threaded implementation, and should be circumvented by using Map reduce.

2. (4) Test of k-mer sizes
   Date: 24 April 2015
   Goal: To find which k-mersize is closest to the gold standard.
   Procedure: I shall use 5 fasta files with approximately 10000 sequences in it to perform tests on. This is because my calculation of the gold standard will be using levenshtein distance which is very slow and therefore the test files cannot be larger than this.

   When the gold standards are found, i shall perform tests using my prototype. These will be by performing tests for 10 different k-mer sizes and 5 different number of hash functions, and see whether there is a pattern.