Max-Minhash Clustering Algorithm of RNA sequences

Mehdi Asser Husum Nadif

May 25, 2015

Contents

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 $h : U \to [r]$ 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

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

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[11], 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 $H = \{h : U \to [r]\}$, H is said to be universal if $\forall x \forall y \in U$

$$\delta_H(x, y) \leq \frac{|H|}{m}$$

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 $h^C_{a,b} : [U] \rightarrow [r]$,

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

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.[6]

Also commonly refered to as **multiply-shift**, this state of the art scheme described in [6] 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[11].

Take a universe $U \geq 2^k$ which is all k-bit numbers. For $l = \{1, .., k\}$, the hash functions $h^D_a(x) : \{0, ..., 2^k - 1\} \rightarrow \{0, ..., 2^l - 1\}$ 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 [13, 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 $x \neq y \pmod{2^k}$. For a proof of this, consult [13][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 and Maxwise Hashing

Minwise hashing, as described in [2], 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 [9], however the method of this paper will be extended by applying two methods of maxwise hashing as described in [7].

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. In contrast to calculating the Hamming Distance or the Levenshtein distance[2], minwise reduces the number of operations needed for the calculation of the Jaccard similarity, by taking advantage of the properties of minwise independent sets[2, pp. 3]. This property will be described below, as well as its application.

Min-wise Independency

Let $H : U \to [r]$ be a class of hashfunctions. Then for any set $X \subseteq [U]$ 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.

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 $h_{\min,i}, i = 1, .., nh$ such that

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

[2]two popular distance metrics that have high precision, but demand long computation time

We may then compute the similarity of two sets $\hat{S}_1$ and $\hat{S}_2$ defined by the above equation as

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

where

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

which is called the **minwise sketch**. As we see from Eq. 2, 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[3] is found in [12], shows 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 [7]. 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 that a set X is said to be maxwise independent if

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

for any $x \in X$. 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 will function almost like the minwise sketch. It is first when combining the two sketches that they have interesting properties.

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 [7], where they halve the amount of hashfunctions, so that for $i = 1, .., k/2$

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

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

Let this method be called **Max-minwise halved sketch** (abbr. $\mathbf{Mm}\frac{1}{2}$). This method has been proven to be double as quick as the min-wise hashing, without loss of precision[7]. It is also shown in Lemma 2 in [7] 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

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

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** (abbr. **Mm**). The expected value of **Mm** is also the Jaccard similarity by the following proof:

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

the final three 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 $\mathbf{Mm}\frac{1}{2}$ and **Mm** is that the first runs only half as many times as the second for each comparison between two sets.

Additional Tools

A few tools remain to be explained before we can jump into the algorithm. These tools have a variety of reasons for being used that will be explained individually.

k-mer

In order to create the sets that $\mathbf{Mm}\frac{1}{2}$ and **Mm** sketches are built with, we shall be using k-mer to partition each sequence into subsets. The k-mer of a sequence string are defined as follows:

The k-mer of a sequence string s is the set of all the substrings of size k of s.

The 1-mer of a sequence string, will therefore be the set of all characters in the sequence string. It is therefore sensible to consider the size of k when partitioning the sequence string.

k-mer transformation

Given that the k-mers will be used as hash function input, it is reasonable to transform them to an input that is easy to map. As sequence strings only comprimise of 4 characters (A,C,G,T), we assign each character a 2 bit value so that $A = \mathbf{00}, C = \mathbf{01}, G = \mathbf{10}, T/U = \mathbf{11}$, and then put them in sequence according to their position in the substring, eg.

```
A G T A C
0010110001
```

which also greatly reduces the memory usage.

MapReduce

Given a sizeable amount of sequences per file, it was quite essential to have a parallel and distributed programming model. For this purpose, MapReduce is a popular programming model. It works by distributing its task to a multitude of workers. Worker can be computers or cores. It expresses its computation as two functions:

1. Map: Runs a function over each element of a list and returns an intermediate value.
2. Reduce: Merges the intermediate values to form a potentially smaller set of values.

As explained in [5], MapReduce processes input by the following steps

1. Map step: Splits the input into M splits. Each split is then distributed to a worker who will perform a Map function on the given split and saves the result into a temporary storage.
2. Shuffle step: The results from the Map calls are then written to a local disk, partitioned into R regions.
3. Reduce step: For each region, a worker is set to run a Reduce job on it, in parallel.

MapReduce has been shown to scaler better than other parallel programming tools for input sizes that surpass 100 Mb, which is why it chosen.[10] Apache Hadoop MapReduce was used, as it is free source. However, since the startup time of Hadoop MapReduce is around 4 seconds, MapReduce is directly slower at small input files.

Pig

Pig Latin is a high level language for compiling and executing MapReduce jobs over Hadoop. Advantageous when performing a pipeline of MapReduce jobs[8], it also demands very few lines of code compared to Hadoop MapReduce code. Additionally, users may write User Defined Functions (abbr. UDF), completely eliminating the need to write any map or reduce function, as they are on the lower level.

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 each.

To perform these tests, I wrote a short java program. It randomized a given number of k-mer transformations[4], 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 $2 \cdot k - 32$.

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.

The results of the tests can be seen in Table 1. The Carter Wegman implementation is consequently slower 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. The differences in speed are however 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 time.

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.

Implementation

In the following section we will describe how each step of the algorithm is performed in pseudo code. Then, in order to describe the MapReduce flow, we shall explore the Pig script written for the algorithm.

Algorithm

The algorithm will take a few input parameters from the user before running

- a list of sequences $S = s_0, s_1, ..., s_{N-1}$ of length N
- k-mer size k.
- H, the number of hash functions in the sketch

[4] which are 2·k bits long

- ϵ, the threshold for similarity.

Initially, the sequences had to be transformed into meaningful input for the hash functions used to produce the sketches **MM** and $\mathbf{MM}\frac{1}{2}$. Natural numbers fit the purpose well, and are exactly what k-mer transformations constitute. The function `kmerTransformation`(s, k), as seen in Alg. 1, is how the k-mer were transformed. `transformChar` applies the same bit-wise transformation as described in the Tools section.

Algorithm 1 Transforms sequence s into its k-mer transformation

Input: a sequence s, a k-mer size k
Functions: `transformChar`(s, k) maps the characters of a k-mer to 2-bit values, `generateKmer`(s, k) returns the k-mer of s.
Output: The transformed k-mer in a list kmerTransformed.

```
1:  function kmerTransformation(s, k)
2:      kmer ← generateKmer(s, k)                          ▷ saves k-mer of s
3:      for gram in kmer do
4:          prod ← 1
5:          sum ← 0
6:          for i ← 0 to gram.length − 1 do
7:              sum ← sum + prod · transformChar(gram.charAt(i))
8:              prod ← prod · 4
9:          kmerTransformed.append(sum)
10:     return kmerTransformed
```

Once the transformations were done, we needed to produce **MM** and $\mathbf{MM}\frac{1}{2}$. As we concluded in the Hash Performance Test, we were to use the Carter Wegman hash function (Eq. 3), which are referred to as $h_i(x)$ from now on. For the sketch, we generate H hashfunctions for **MM** and $H/2$ hash functions for $\mathbf{MM}\frac{1}{2}$, each with a different a and b. They take the form

$$h_i(x) = ((a_i \cdot x + b_i) \mod p) \mod 10000$$

where $a = i + 1$,$b = i$, and $p = 1845587707$ for $i = 0, 1, ..., H - 1$. The 10000 at the end was to assert that the change of collision was 0.01%, a sufficient precision for the purpose.

Alg. 2 shows how **MM** was produced. As we can see, it runs through all H hash functions to find the value that is smallest and highest in the transformed k-mer list. Alg. 3 shows the pseudocode for calculating $\mathbf{MM}\frac{1}{2}$, where the only change needed was to divide the number of hash functions by two.
The sketches ready, what remained was to compare all sequences' sketches to each other. As speed was the main goal, a greedy algorithm was developed for both **MM** and $\mathbf{MM}\frac{1}{2}$.

The greed lays in that if the similarity between two sequences is above ϵ, they are immediately placed in the same cluster, regardless of whether there are other clusters that the other sequence is more similar to.

Algorithm 2 Uses transformed k-mer to find **MM** of a sequence

Input: a list of transformed k-mer kT of a sequence, H number of hashfunctions.
Parameters: $h_i(x)$ the ith hash function of the H hash functions initiated.
Output: **MM** of a sequence.

```
1:  function toMMSketch(kT)
2:      MM ← Ø
3:      for i ← 0 to H − 1 do
4:          hmin ← ∞
5:          hmax ← −∞
6:          for kt in kT do
7:              val ← h_i(kt)
8:              if val < hmin then hmin = val
9:              if val > hmax then hmax = val
10:         MM.append([hmin,hmax])
11:     return kmerTransformed
```

Algorithm 3 Uses transformed k-mer to find $\mathbf{MM}\frac{1}{2}$ of a sequence

Input: a list of transformed k-mer kT of a sequence, H number of hashfunctions.
Parameters: $h_i(x)$ the ith hash function of the $H/2$ hash functions initiated.
Output: $\mathbf{MM}\frac{1}{2}$ of a sequence.

```
1:  function toMMSketch(kT)
2:      MMhalf ← Ø
3:      for i ← 0 to H/2 − 1 do
4:          hmin ← ∞
5:          hmax ← −∞
6:          for kt in kT do
7:              val ← h_i(kt)
8:              if val < hmin then hmin = val
9:              if val > hmax then hmax = val
10:         MMhalf.append([hmin,hmax])
11:     return kmerTransformed
```

Alg. 4 shows the algorithm for **MM** greedy clustering. In line 1-7 the sketches are prepared for the comparison; the remainder is the greedy algorithm. The most interesting part of this algorithm is line 15-22, where the similarity between two strings is determined. Running through all H hash functions, in line 16-20 it checks: if either $h_{min,i}$ or $h_{max,i}$ of both sequences are equal to the other sequence's counterpart, an intersection occurs. When all hash functions have been run through, the similarity between the two strings is determined in line 21 as the intersections over the number of hash functions. If the similarity then surpasses the given ϵ the strings are placed in the same cluster.

Alg. 5 has a very similar method as Alg. 5. The main difference lies in line 15-22, where the intersections are determined by the number of $h_{min,i}$ that are equal plus the number of $h_{max,i]}$ that are equal. This means that each hash function can result in two intersections, instead of one in Alg. 4. Line 20 the shows the similarity measure, which is the same as in Alg. 4, since the potential number of intersections per hash function was doubled.

To render more intelligible the difference between the two algorithms, let us analyze the order of growth of both algorithms.

Order of Growth: MM

Each line is analyzed in Alg. 4.
line 5: `kmerTransformation` for a string of length l, loops $(l-k+1)$ times through the string to produce all k-mer. Each k-mer is then iterated over k times. This gives a $\Theta((l-k+1)k)$.
line 6: `toMMSketch` for a string of length l, repeats H times $(l-k+1)$ comparisons. This gives $\Theta((l-k+1)H)$.
line 3 - 6: Together, the lines 5 and 6 are repeated N times, giving a runtime of $\Theta(N(l-k+1)(k+H))$.
line 8 - 22: Let us first analyse the lower bound. If we assume all sequences will be assigned to the same cluster, line 8 will be run once. Line 12 will be run N times, and line 15 will be run H times per iteration of line 12. The lower bound will therefore be $\Omega(N \cdot H)$. In worst case, all string belong in different clusters. In this case, line 8 will be run N times, line 12 N times per iteration of line 8, and line 15 H times per iteration of line 12. This gives an upper bound of $O(N \cdot N \cdot H)$.

Conclusively, the runtime will be $\Theta(N(l-k+1)(k+H))+O(N \cdot N \cdot H)$. Assuming that $l = 1500$[5], we would need a $N > 1500$ to conclusively say that the runtime is $O(N \cdot N \cdot H)$.

Order of Growth: MM$\frac{1}{2}$

As the analysis is completely parallel to the one performed for **MM** immediately above, we shall simply conclude that the runtime will be $\Theta(N(l-k+1)(k+H/2))+O(N \cdot N \cdot (H/2))$.

[5]the maximal length of sequences we wish to test our program on.

Algorithm 4 Greedy Clustering using **MM**

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, ..., s_{N-1}\}$ of length N.
Parameters: H number of hash functions, k size of k-mer, ϵ threshold for string similarity, c current cluster
Output: A list of the cluster C each sequence belongs to.

Initialize $h_i(x)$ for $i = 0, 1, ..., H$
Initialize kT,C, and MM as lists of length N
for $i \leftarrow 0$ to $N - 1$ **do**
    C[i] $\leftarrow 0$
    kT[i] $\leftarrow$ kmerTransformation(s_i)
    MM[i] $\leftarrow$ toMMSketch(kT[i])
$c \leftarrow 0$
for $i \leftarrow 0$ to $N - 1$ **do**
    if C[i] == 0 **then**
        $c \leftarrow c + 1$
        C[i] $\leftarrow c$
        for $j \leftarrow 0$ to $N - 1$ **do**
            $intersections \leftarrow 0$
            if C[i] == 0 **then**
                for $k \leftarrow 0$ to $H - 1$ **do**
                    if MM[i][k][0] == MM[j][k][0] **then**
                        $intersections \leftarrow intersections + 1$
                    else
                        if MM[i][k][1] == MM[j][k][1] **then**
                            $intersections \leftarrow intersections + 1$
                if $\frac{intersections}{H} \geq \epsilon$ **then**
                    C[j] $\leftarrow C$[i]

Algorithm 5 Greedy Clustering using $\mathbf{MM}^{\frac{1}{2}}$

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, ..., s_{N-1}\}$ of length N.
Parameters: H number of hash functions, k size of k-mer, ϵ threshold for string similarity, c current cluster
Output: A list of the cluster C each sequence belongs to.

Initialize $h_i(x)$ for $i = 0, 1, ..., H/2$
Initialize kT,C, and MM as lists of length N
for $i \leftarrow 0$ to $N-1$ **do**
    C[i] $\leftarrow 0$
    kT[i] $\leftarrow$ kmerTransformation(s_i)
    MM[i] $\leftarrow$ toMMhalfSketch(kT[i])
$c \leftarrow 0$
for $i \leftarrow 0$ to $N-1$ **do**
    if C[i] == 0 **then**
        $c \leftarrow c + 1$
        C[i] $\leftarrow c$
        for $j \leftarrow 0$ to $N-1$ **do**
            $intersections \leftarrow 0$
            if C[i] == 0 **then**
                for $k \leftarrow 0$ to $H/2 - 1$ **do**
                    if MM[i][k][0] == MM[j][k][0] **then**
                        $intersections \leftarrow intersections + 1$
                    if MM[i][k][1] == MM[j][k][1] **then**
                        $intersections \leftarrow intersections + 1$
                if $\frac{intersections}{H} \geq \epsilon$ **then**
                    C[j] $\leftarrow C$[i]

MapReduce Framework

In order to parallellize and distribute the algorithm, MapReduce and Pig were used. Fig. 1 shows the flow of this setup is. As we can see, the fasta file is first dealt out, so that each sequence is assigned to a worker. Then, each sequence has the transformation from sequence to sketch performed in parallel. When this transformation is done, all the sequences are passed to the Greedy Clustering algorithm at once, were the clusters are found. Finally, this is saved into an output.

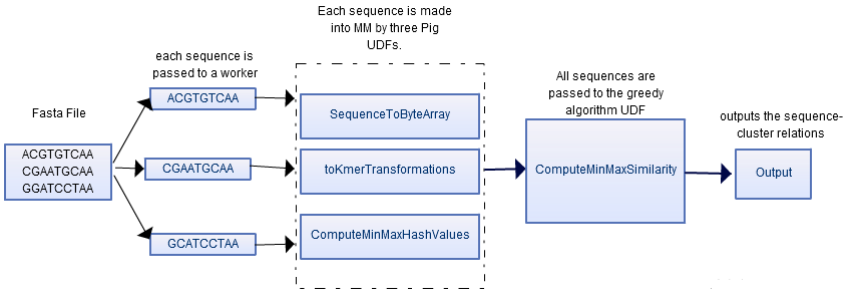

Figure 1: The flow of the MapReduce FrameWork using Pig

The Pig Script used to create this flow for the **MM** algorithm can be seen in Alg. 6. Line 1 loads all the sequences from the file. Then, in line 2 the sequences are converted to bytes, which facilitates the conversion of the k-mer. The $PROGRAM variable defines whether the input is DNA or RNA. In line 3, all files that contained an unknown character are sorted out. Line 4 produces the k-mer transformations for k-mer size $KMER_SIZE. The **MM** sketch is then produced in line 5, using $HASH_SIZE hash functions. Line 6 and 7 are used to count the number of sequences to cluster. Finally, line 8 takes all sequences in to run the Greedy Clustering on them. ϵ is given by $THRESHOLD. In line 8 MapReduce is not applied, as it runs the function on all sequences at once. So, the advantage of MapReduce over a non-parallel solution will only affect line 1 through 7.

Dataset Description

Algorithm 6 Pig Script for MapReduce **MM** Greedy Cluster Algorithm

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, ..., s_{N-1}\}$ of length N.
Parameters: $HASH_SIZE number of hash functions, $KMER_SIZE size of k-mer, $THRESHOLD is ϵ
Output: list of the cluster each sequence belongs to in **results**.

```
1: sequences = LOAD '$INPUT' USING FastaReader AS (line: chararray);
2: converted = FOREACH sequences GENERATE SequenceToByteArray(line, $PROGRAM);
3: NoN = FILTER converted BY byteSeq is not null;
4: kmers = FOREACH NoN GENERATE toKmerTransformations(byteSeq, $KMER_SIZE);
5: minmaxvalues = FOREACH kmers GENERATE FLATTEN (ComputeMaxMinHashValues (kmerlist, $HASH_SIZE, $KMER_SIZE));
6: grouped = GROUP minmaxvalues ALL;
7: a_count = FOREACH grouped GENERATE COUNT (minmaxvalues);
8: results = FOREACH grouped GENERATE FLATTEN (ComputeMaxMinSimilarity (minmaxvalues, a_count.$0, $HASH_SIZE, $THRESHOLD));
```

Precision Tests

Here are the diagrams for **Mm**

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] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] 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.

[7] 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.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[9] 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.

[10] R. Stewart and J. Singer. Comparing fork/join and mapreduce. 2013.

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

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

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