Max-Minhash Clustering Algorithm of RNA sequences

Mehdi Asser Husum Nadif

June 1, 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 $x \neq y$ (mod 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, a table of random $h_{\min,i}, i = 1, .., nh$ is produced, 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

Given a set S_2 with a $\hat{S_2}$, the similarity of S_1 and S_2 can then be defined by

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

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**. The influence the size of nh will have on the error can be proven. Using Chernoff Bounds[3], [12] proves that the relation between nh and ϵ, the error, is

$$\epsilon = O\left(\frac{1}{\sqrt{nh}}\right) \tag{10}$$

Thus, an $nh = 100$ should give 10% error. The necessary number of hash functions for this error can however be halved by making a few slight modifications.

Max-wise hashing

The aforementioned modification of the minwise sketch 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, the maxwise independent set is appended. Very literally, this means that instead of using the minimum hashvalue, the maximal hashvalue are used such that a set X is said to be maxwise independent if

$$\Pr(h_{\max}(X) = h(x)) = \frac{1}{|X|}, h_{\max} = \max\{\forall x \in X, h(x)\} \tag{11}$$

for any $x \in X$. The Jaccard similarity measure for two sets A and B is

$$\Pr(h_{\max}(A) = h_{\max}(B)) = \frac{|A \cap B|}{|A \cup B|} \tag{12}$$

and finally for a random set S_1, given a table of random $h_{\max,i}, i = 1, ..., nh$, a sketch can be made, like so

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

This sketch is called the maxwise sketch, and functions almost like the minwise sketch. It is first when combining the maxwise- and minwise sketch that they have interesting properties.

Combining Max-wise and Min-wise

There are two ways of combining the max-wise and the min-wise sketches. One is the method in [7], where they halve the amount of hashfunctions, so that for $i = 1, .., nh/2$

$$J(A, B) = \frac{1}{nh} \sum_{i=1}^{nh/2} (h_{\min,i}(A) = h_{\min,i}(B) + h_{\max,i}(A) = h_{\max,i}(B)) \tag{13}$$

[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 $i = 1, .., nh/2$

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

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

$$J(A,B) = \frac{1}{nh} \sum_{i=1}^{nh} (h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B)) \tag{14}$$

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:

$$\begin{aligned} &\frac{1}{nh} \sum_{i=1}^{nh} (h_{\min,i}(A) = h_{\min,i}(B) | h_{\max,i}(A) = h_{\max,i}(B)) = \\ &\frac{1}{nh} \sum_{i=1}^{nh} (J(A,B) | J(A,B)) = J(A,B) | J(A,B) = J(A,B) \end{aligned} \tag{15}$$

the final three steps follow from Eq. 9 and Eq. 12.It follows that this method also calculates 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. The error of these functions can similarly be found to be

$$\epsilon = O\left(\frac{1}{\sqrt{2nh}}\right) \tag{16}$$

meaning that $nh = 50$ would give an error of 10%.

Additional Tools

A few tools remain to be explained before the algorithm can be described. 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, k-mer will be used 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), each character is assigned 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.

Levenshtein Distance

For a precise distance metric between two string, the levenshtein distance is quite applicable. Given two strings, s_1 and s_2, the **Levenshtein distance** is defined as the minimum number of single character edits necessary to change s_1 into s_2. The **Levenshtein similarity** could then be calculated as

$$sim = \frac{\max(s_1, s_2) - \text{LevenshteinDistance}(s_1, s_2)}{\max(s_1, s_2)} \tag{17}$$

where $\max(s_1, s_2)$ is the maximal length of s_1 and s_2.

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

[4]which are 2·k bits long

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, each step of the algorithm will be described using pseudo code. Then, in order to describe the MapReduce flow, the Pig scripts written for the algorithms are analyzed.

Algorithm

The algorithm takes 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
- 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, a function for producing **MM-** and $\mathbf{MM}\frac{1}{2}$ sketches should be made. As concluded in the Hash Performance Test, the

Carter Wegman hash function (Eq. 3) was most applicable. These hash functions will be referred to as $h_i(x)$ from now on. For the sketch, H hashfunctions are generated 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. 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.

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.

```
function toMMSketch($kT$)
    MM ← Ø
    for $i$ ← 0 to $H - 1$ do
        hmin ← ∞
        hmax ← −∞
        for $kt$ in $kT$ do
            val ← $h_i(kt)$
            if val < hmin then hmin = val
            if val > hmax then hmax = val
        MM.append([hmin,hmax])
    return kmerTransformed
```

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.

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

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.

```
function toMMSketch($kT$)
    MMhalf $\leftarrow \emptyset$
    for $i \leftarrow 0$ to $H/2 - 1$ do
        hmin $\leftarrow \infty$
        hmax $\leftarrow -\infty$
        for $kt$ in $kT$ do
            val $\leftarrow h_i(kt)$
            if val $<$ hmin then hmin = val
            if val $>$ hmax then hmax = val
        MMhalf.append([hmin,hmax])
    return kmerTransformed
```

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. Assuming 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], it would necessitate an $N > 1500$ 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, the runtime will be $\Theta(N(l-k+1)(k+H/2))+O(N\cdot N\cdot(H/2))$, halving the number of hash functions needed.

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.

```
 1: Initialize $h_i(x)$ for $i = 0, 1, ..., H$
 2: Initialize $kT$,$C$, and $MM$ as lists of length $N$
 3: for $i \leftarrow 0$ to $N - 1$ do
 4:     $C$[i] $\leftarrow 0$
 5:     $kT$[i] $\leftarrow$ kmerTransformation($s_i$)
 6:     $MM$[i] $\leftarrow$ toMMSketch($kT$[i])
 7: $c \leftarrow 0$
 8: for $i \leftarrow 0$ to $N - 1$ do
 9:     if $C$[i] == 0 then
10:         $c \leftarrow c + 1$
11:         $C$[i] $\leftarrow c$
12:         for $j \leftarrow 0$ to $N - 1$ do
13:             $intersections \leftarrow 0$
14:             if $C$[i] == 0 then
15:                 for $k \leftarrow 0$ to $H - 1$ do
16:                     if $MM$[i][k][0] == $MM$[j][k][0] then
17:                         $intersections \leftarrow intersections + 1$
18:                     else
19:                         if $MM$[i][k][1] == $MM$[j][k][1] then
20:                             $intersections \leftarrow intersections + 1$
21:                 if $\frac{intersections}{H} \geq \epsilon$ then
22:                     $C$[j] $\leftarrow C$[i]
```

[5]the limit of length of sequences the program shall be tested on.

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. The fasta file is first partitioned, 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, where the clusters are found. Finally, this is saved into an output.

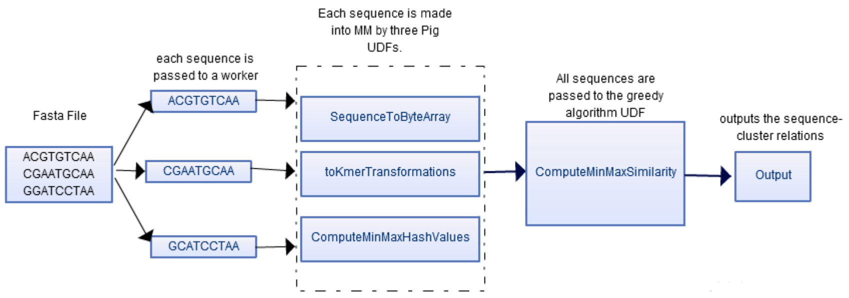

Each sequence is made into MM by three Pig UDFs.
each sequence is passed to a worker
Fasta File
ACGTGTCAA
CGAATGCAA
GGATCCTAA
ACGTGTCAA
CGAATGCAA
GCATCCTAA
SequenceToByteArray
toKmerTransformations
ComputeMinMaxHashValues
All sequences are passed to the greedy algorithm UDF
ComputeMinMaxSimilarity
outputs the sequence-cluster relations
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.

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));
```

Test Data

For the experiments, it was important that the data used was representative of real life data sets. Therefore, all data used were DNA and RNA sequences extracted from bacteria and biological tissues. As we shall perform two different sets of experiments, each will have a seperate data set for tests; these will be described below.

Data used for precision tests

In order to test the precision of both algorithms, it was necessary to have relatively small sets, since we had to calculate a gold standard[6]. We used a clean sample of Cecum DNA data consisiting of 81.029 sequences. These were divided into 5 seperate sample files, each of which contained a sample of 10.000 sequences. These files will be refered to as `Cecum1`, `Cecum2`, `Cecum3`, `Cecum4`, and `Cecum5`. Each file contained a unique sample of the original file, none of which intersect with any of the other files.

Data used for speed tests

For the speed tests, two seperate files were taken in use, one DNA and one RNA. The DNA sample was of uncultured Actinobacteria, consisting of 2.879.170 sequences. This file was divided into five samples for the speed tests.

1. `Actino50K`: A sample of 50.000 sequences. Size = 72.5 MB
2. `Actino100K`: A sample of 100.000 sequences. Size = 146.6 MB
3. `Actine200K`: A sample of 200.000 sequences. Size = 283.5 MB
4. `Actino500K`: A sample of 500.000 sequences. Size = 663.4 MB

[6]which is a very costly affair

5. `Actino1mio`: A sample of 1.000.000 sequences. Size = 1286 MB

The SILVA SSU 119[7] database was taken into use as the RNA sample for the speed tests. It consists of all aligned sequences with a high alignment identity, without any sequences of 99% similarity. Just as the DNA sample, five samples from this file were used for testing

1. `Silva50K`: A sample of 50.000 sequences. Size = 80 MB
2. `Silva100K`: A sample of 100.000 sequences. Size = 160 MB
3. `Silva200K`: A sample of 200.000 sequences. Size = 320 MB
4. `Silva500K`: A sample of 500.000 sequences. Size = 790.5 MB
5. `Silva1mio`: A sample of 1.000.000 sequences. Size = 1573 MB

All samples consist of clean sequences, meaning they only contain the four characters DNA or RNA consist of. As one may note, the RNA samples are bigger than the DNA samples. This is caused by the fact that the sequences are longer in the RNA samples. These two files were chosen to see how the size of the sequences would affect the speed of the **MM** and $\mathbf{MM}\frac{1}{2}$ algorithms compared to `uClust`.

Hardware

A Lenovo IdeaPad Y500 laptop was taken in use for the tests. Its processor is a Quad-Core 2.4 GHz, it has 8 GB RAM, and finally a NVIDIA GeForce GT 650M graphics card.

Experiments

To access efficiency of both algorithms, and their MapReduce frameworks set of experiements was performed on the data just described. Two different forms of tests were performed; some for measuring the precision of the algorithms, and some for measuring the speed of the algorithms.

For all experiments, $\epsilon = 0.95$. The **MM** clustering algorithm will be refered to as **MM**, likewise will the $\mathbf{MM}\frac{1}{2}$ clustering algorithm be referred to as $\mathbf{MM}\frac{1}{2}$.

Precision tests

In Eq. 16, a metric of the error was set up for the number of hash functions H. This error did not provide an answer to relation between H and the size of k-mers k and its error. Therefore, a practical testing of this relation was performed, to investigae the precision of **MM** and $\mathbf{MM}\frac{1}{2}$.

A gold standard was instrumental to determine the precision of our algorithms. For this purpose, the Levenshtein similarity was used, as defined in the Tools section. If two strings had a Levenshtein similarity above ϵ they were placed in

[7]http://www.arb-silva.de/documentation/release-119/

the same cluster. The gold standard, **GS**, were then set to be the number of clusters found by the Levenshtein similarity clustering algorithm. Once the **GS** were retrieved, the error of an algorithms' found number of clusters, C, could be defined as the difference between C and the gold standard **GS**

$$E = |C - \mathbf{GS}|$$

similarly, the percentage of error E_p was

$$E_p = \frac{|C - \mathbf{GS}|}{\mathbf{GS}}$$

The lower E was, the more precise the algorithm. As both **MM** and $\mathbf{MM}\frac{1}{2}$ results depend on k and H, a large set of tests were performed to determine the precision of each algorithm at many k and H. When the optimal settings were found, the results could then be compared to `uClust` algorithm's precision.

MM precision tests

First, the **MM** algorithm's precision tests were made. In order to narrow down the number of k and H to test, a heat plot of a wide set of k and H for seeing the big scope was made, as seen in Fig. **??**. From this wide study, ostensibly the error increased rapidly at $k > 13$, and $k < 5$ were less precise than $k = 5$. Also, at $H > 80$ the error seemed to stop changing significantly.

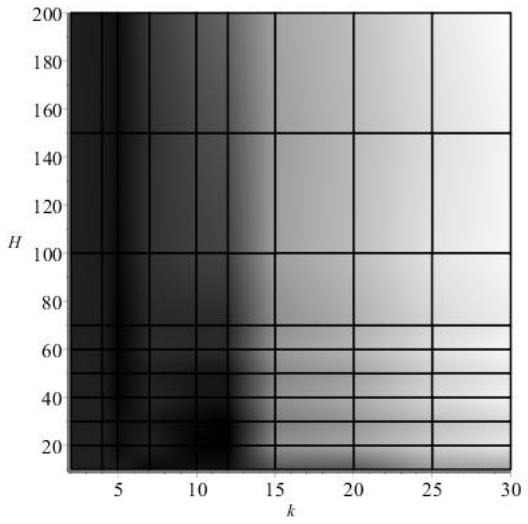


Figure 2: Heat map of the error of **MM** at a widely dispersed array of k and H, run on `Cecum1`. Darkness increases as the error decreases.

The results of this test showed that the more rigourous tests could be performed at a more narrow set of k and H, as seen in Fig. 3. All heat maps showed very similar results for all samples. There were specific areas that seemed to have repeatedly have a very low error, specifically at

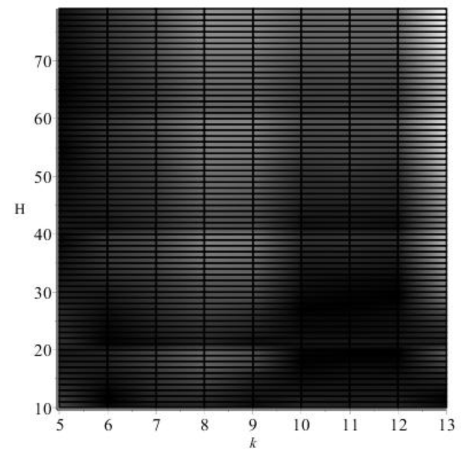
- k=5, a black line from $H = 50$ to $H = 80$.

- $k = 6$, two black spots at around $H = 10$ and $H = 22$.
- $k = 10, 11, 12$, two big black spots at around $H = 16$ and $H = 30$.

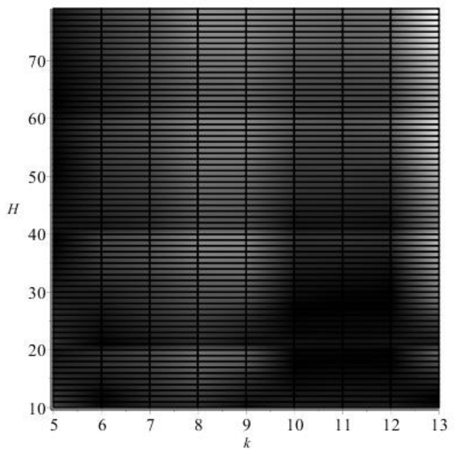
These areas of interest were deemed to be most precise. Therefore, each was tested alongside `uClust`, to be able to compare them to each other. Fig. 4 shows the results of these tests.

Fig. 4(a) shows that at $k = 5$, the overall performace of the algorithm appeared near comparable to `uClust`, with a maximum error of approx. 33%. At the spikes, the precision is even higher than `uClust`, most notably at $H = 54$, $H = 65$ and $H = 66$, where the average error over all samples was around 4%. Fig. 4(b) shows that at $k = 6$, the maximal error is around 70%, much worse than that of `uClust`. However, two spikes also appeared here at $H = 13$ and $H = 22$, both with an average error of around 7%, slightly better than `uClust`.

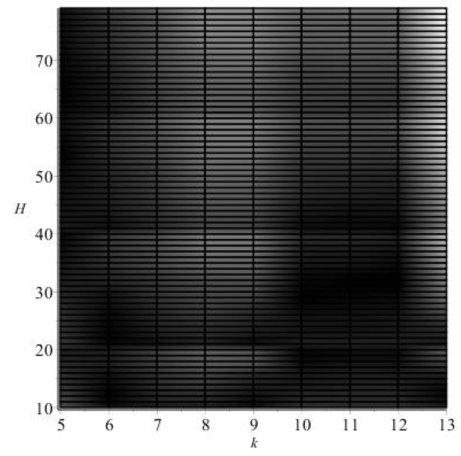
Fig. 4(c-e) show that in spite of generally portraying a high error overall at $k = 10, 11, 12$, reaching maximum errors of over 80%, they still had potentially low errors; Most notably at $k = 10, H = 20$, $k = 11, H = 30$, and $k = 12, H = 30$, with spikes with an average error about 10%, about the same as `uClust`.



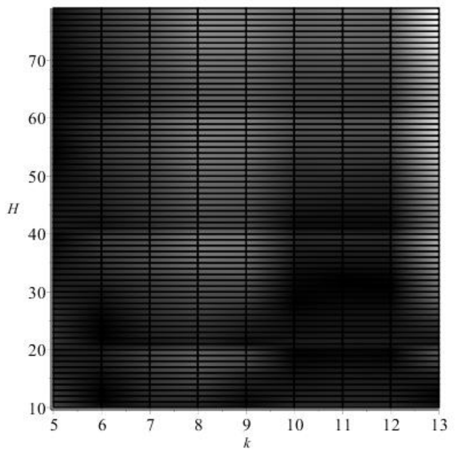
(a) `Cecum1`



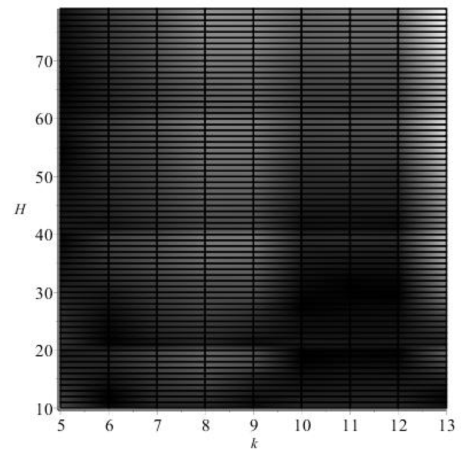
(b) `Cecum2`



(c) `Cecum3`

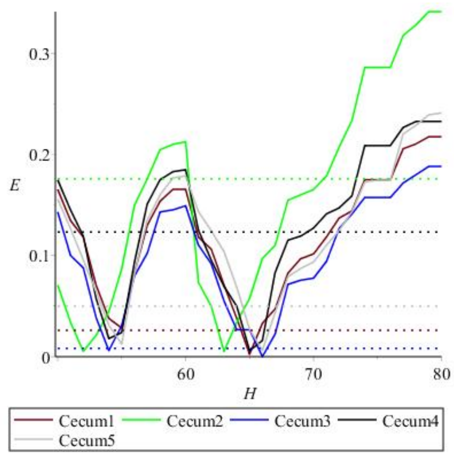


(d) `Cecum4`

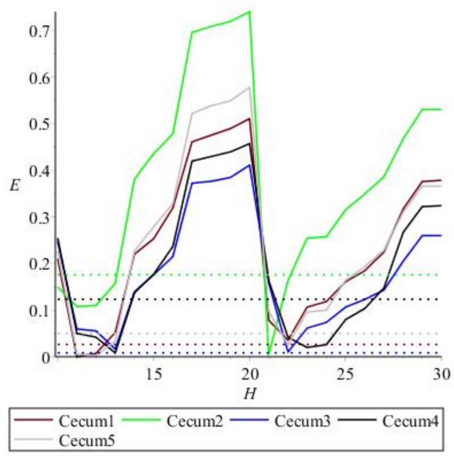


(e) `Cecum5`

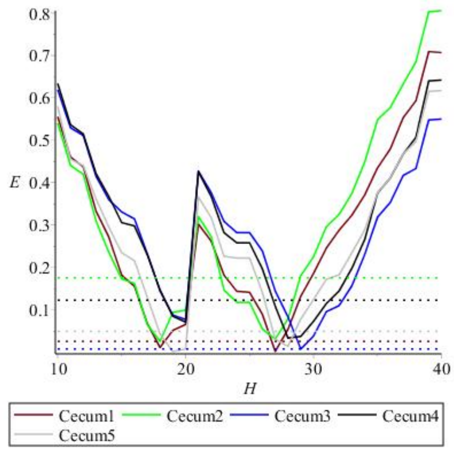
Figure 3: Heat maps of the error of **MM** at a narrow array of k and H at all 5 samples of `Cecum` DNA. Darkness increases as the error decreases.



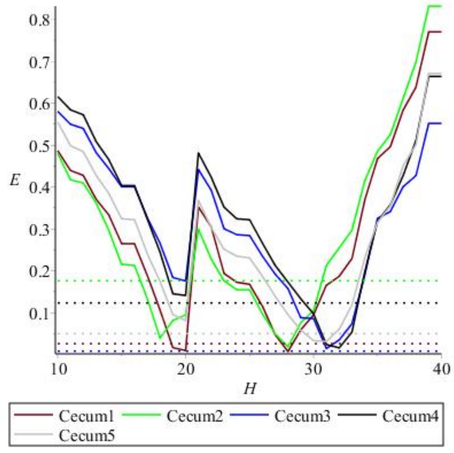
(a) `k=5`



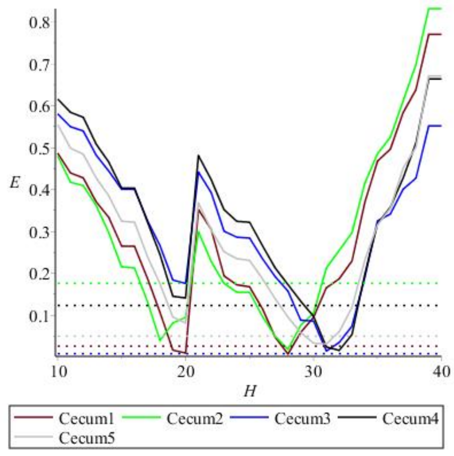
(b) `k=6`



(c) `k=10`



(d) `k=11`



(e) `k=10`

Figure 4: Plots with H on the x-axis and error E_p in percent ($0.5 = 50\%$) the y-axis, each plot at different k. The lines in the plots signify the error at the given k and H of **MM**, while the dotted lines signify the error of `uClust` at each file. The colours define the sample file.

$\mathbf{MM}\frac{1}{2}$ precision tests

The tests of $\mathbf{MM}\frac{1}{2}$ were performed completely parallel to those of **MM**, but with lower H as reputedly $\mathbf{MM}\frac{1}{2}$ only needs half as many hash functions as the minwise sketch[8]. In Fig. 5 figures the test of a wide array of k and H.

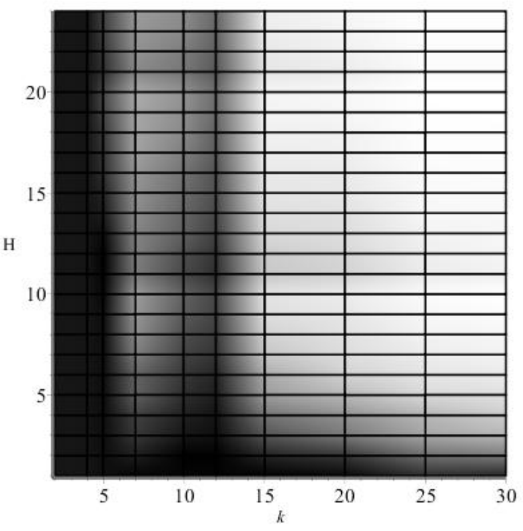


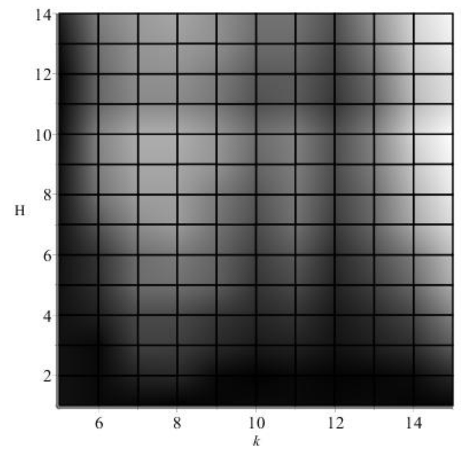
Figure 5: Heat map of the error of $\mathbf{MM}\frac{1}{2}$ at a widely dispersed array of k and H, run on `Cecum1`. Darkness increases as the error decreases.

The large white plateau in Fig. 5 meant that $k > 15$ could be ignored, as the error at this area was very high. Similarly, $k < 5$ gave more error than $k = 5$ and could therefore be ignored. H could be reduced to $H < 15$, as the blackest spot seems around $k = 5, H = 12$.

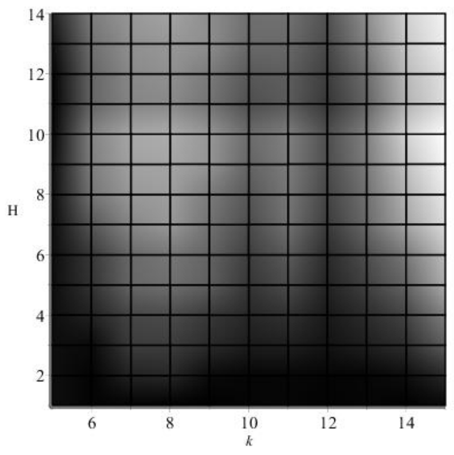
New tests were performed with these limitations. The results can be seen in Fig. 6. All five plot are almost completely identical, showing that mostly at $H = 1$ and $H = 2$, the precision is highest. The only exception was at $k = 5$, where there seemes to be a lower error along all H. An additional analysis showed that the minimum error in each heat map of Fig. 6 was always at $k = 8, H = 1$. This result was but a stroke of luck, since a single hash function would theoretically cause the error of the $\mathbf{MM}\frac{1}{2}$ sketch to be too high to be applicable for proper comparison (see Eq. 16).

For further analysis of the k of interest, the errors of each file at $k = 5$,$k = 8$ and $k = 12$ were plotted, as seen in Fig. 7. Fig. 7(a-c) all have maximum errors that surpass 100%, even reaching as high as 700% error. Only in Fig. 7(a) a cuspidate spike was present at $H = 12$ with an average error of 20%, around the double of `uClust` average error.

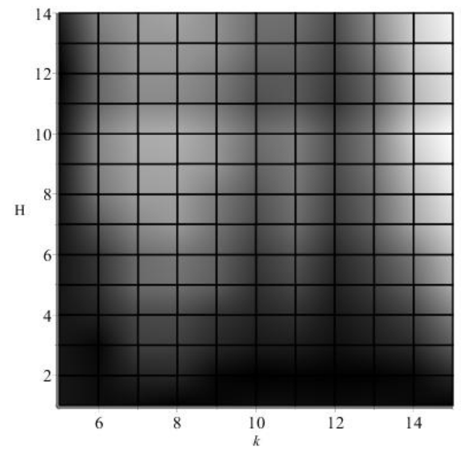
[8]see Minwise and Maxwise Hashing Section



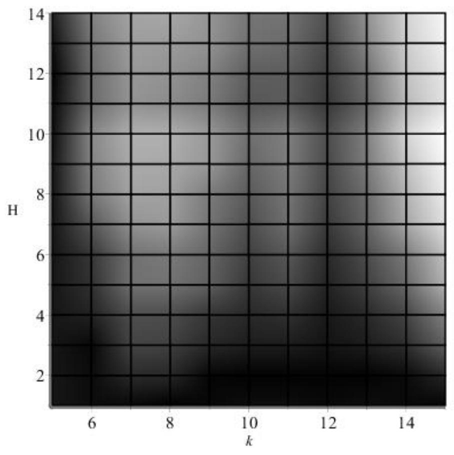
(a) `Cecum1`



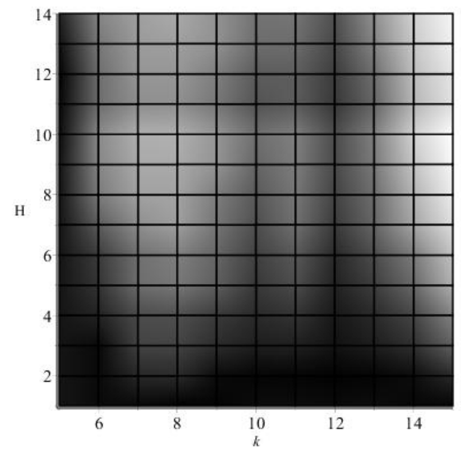
(b) `Cecum2`



(c) `Cecum3`

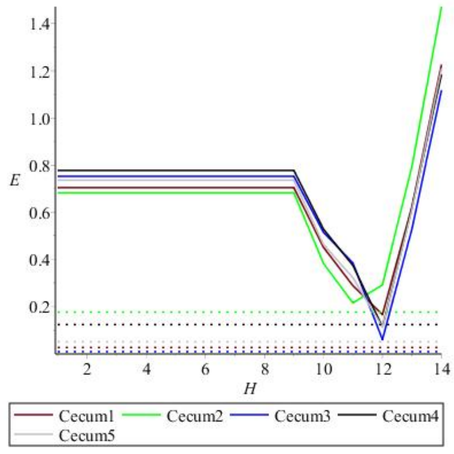


(d) `Cecum4`

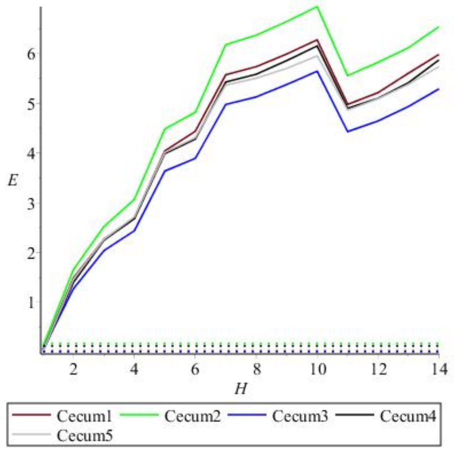


(e) `Cecum5`

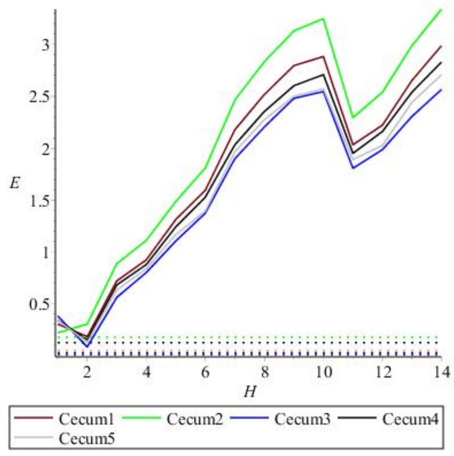
Figure 6: Heat maps of the error of $\mathbf{MM}\frac{1}{2}$ at a narrow array of k and H at all 5 samples of `Cecum` DNA. Darkness increases as the error decreases



(a) `k=5`



(b) `k=8`



(c) `k=12`

Figure 7: Plots with H on the x-axis and error E_p in percent ($0.5 = 50\%$) on the y-axis, each plot at different k. The lines in the plots signify the error at the given k and H of $\mathbf{MM}\frac{1}{2}$, while the dotted lines signify the error of `uClust` at each file. The colours define the sample file.

Comparison

To determine the precision of **MM**, $\mathbf{MM}\frac{1}{2}$ and `uClust`, Fig. 7 and Fig. 4 were perfect candidates. One of the most notable differences between the two graphs was the structure of the results for **MM**, which took an almost "W" like form. The reason for this is from how the sketches of two sequences are compared in Eq. 14. In contrast to Eq. 13, **MM** takes advantage of the strong points of both the max- and minwise hashing by finding the intersection of jaccards. This means that if the maxwise hashing is most precise at $H = 12$, and minwise is most precise at $H = 23$, the spikes will appear at these two spots.

Comparing the two, **MM** was quite simply much more precise than $\mathbf{MM}\frac{1}{2}$, by very large margin. In fact, at the most precise, **MM**$\frac{1}{2}$ at $k = 5$ was still not as good as the worst **MM** was at $k = 5$ in our tests. In fact, **MM** turned out to be even more precise than `uClust` at some very specific parameter settings of k and H.

The extreme imprecision of **MM**$\frac{1}{2}$ was in fact so high, that it proved almost nonfunctional for practical purposes, if precision in any way was considered of any importance. **MM** however proved that it could compete with `uClust` in terms of precision. Therefore, **MM** was considered a good candidate for an alternative to `uClust`. What remained was to determine whether it could compare speed-wise.

Speed tests

For testing the speed, the MapReduce Framework using Pig Scripts were taken in use for **MM** and **MM**$\frac{1}{2}$. `uClust` was run with the setting: `cluster_fast input -id 0.95 -uc output` as storing is also included in the Pig Scripts. As only the 32 bit version of `uClust` was available, there was a memory limit to the size of the operation. For this reason, some of the sample files could not be finished by `uClust`; this occurence will be noted as Mem. Exed.

To achieve the highest speed without loss of too much precision, the precision tests were used to determine the optimal parameters. **MM** was run at $k = 6$ and $H = 13$, which as seen in Fig. 4(b) is comparable to `uClust` in terms of precision. **MM**$\frac{1}{2}$ was run with $k = 5$ and $H = 12$.

The results of the test can be seen in Table 2. As expected, **MM**$\frac{1}{2}$ was consequently faster than **MM**. Theoretically, **MM**$\frac{1}{2}$ should be twice as fast as **MM**. This turned out true in all cases except for the runtime of `Actino1mio` and `Actino500K`. Here **MM**$\frac{1}{2}$ was 10 times faster than **MM**, caused by the fact that fewer clusters were produced by **MM**$\frac{1}{2}$. Therefore the outer loop of the **MM**$\frac{1}{2}$ greedy algorithm iterated less times than the outer loop of **MM**. Such a difference in speed was not predicted, but says more about the imprecision of **MM**$\frac{1}{2}$ than the algorithms' relative speed.

There was a duality of speed between the two sample sets. In the `Actino` sample set, `uClust` consistently ran faster than both **MM** and **MM**$\frac{1}{2}$, with an increasing difference in speed over sample size. In the `Silva` samples, both **MM** and **MM**$\frac{1}{2}$ ran faster than `uClust` at sample size higher than `Silva50K`. In fact, **MM** was almost twice as fast as `uClust` on the run of `Silva500K`, and the difference in speed increased with sample size.

While the frequency of cases where **MM** and **MM**$\frac{1}{2}$ are faster than `uClust` remains unknown, occasionally **MM** and **MM**$\frac{1}{2}$ trumps `uClust`. In choosing between **MM**$\frac{1}{2}$ and **MM**, a strong case could be made for **MM**. **MM**$\frac{1}{2}$ may have been double as quick as **MM**, but this at the cost of precision. So much precision, that one goes from statistically useless using **MM**$\frac{1}{2}$ to competing with `uClust` using **MM**. And since **MM** was also quicker than `uClust` in some cases, it would still serve as a good alternative to `uClust`.

	runtime of sample in s		
Sample	**uClust**	**MM**	**MM$\frac{1}{2}$**
`Actino50K`	2.0	18.3	13.6
`Actino100K`	5.0	23.2	18.7
`Actino200K`	9.0	53.4	28.8
`Actino500K`	14.0	208.4	58.3
`Actino1mio`	Mem. Excd.	1033.8	103.3
`Silva50K`	16.0	18.3	13.6
`Silva100K`	33.0	28.3	18.3
`Silva200K`	70.0	48.6	33.5
`Silva500K`	208.0	133.5	68.3
`Silva1mio`	Mem. Excd.	273.9	128.5

Table 2: Comparison of clustering speed of three algorithms on 10 samples. `uClust` is run with `cluster_fast` option, **MM** with $k = 6$ and $H = 13$, **MM**$\frac{1}{2}$ has $k = 5$ and $H = 12$. The runtime the clustering speed is given in seconds.

Conclusion

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.