1 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.

1.1 Algorithm

The algorithm takes a few input parameters from the user before running

- An N-sized list of sequences $S = s_0, s_1, \ldots, s_{N-1}$, each sequence of variable length.
- 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 max-min-wise sketches. 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 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 character of a k-mer to 2-bit values, `generateKmer`(s, k) returns all k-mers of s.
Output: The transformed k-mer in a list kmerTransformed.

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

Once the transformations were done, a function for producing the max-min-wise sketches for **MM**- and $\mathbf{MM}\frac{1}{2}$ was made. As concluded in the Hash Performance Test, the Carter Wegman hash function (Eq. **??**) 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 the sketch of **MM** and $H/2$ hash functions for the sketch of $\mathbf{MM}\frac{1}{2}$, 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 $i = 0, 1, \ldots, H - 1$. The 10000 at the end was to ensure that the change of collision was 0.01%, a sufficient precision for the purpose.

Alg. 2 shows how the max-min-wise sketch for **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 max-min-wise sketch for $\mathbf{MM}\frac{1}{2}$, where the only change needed was to divide the number of hash functions by two.

Algorithm 2 Uses transformed k-mer to find sketch for **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 sketch for $\mathbf{MM}\frac{1}{2}$ 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 toMMHalfSketch(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
```

As the sketches for **MM** and $\mathbf{MM}\frac{1}{2}$ could now be prepared, what remained was to find the Jaccard similarity coefficient of these sketches. Using **MM** and $\mathbf{MM}\frac{1}{2}$ as the similarity coefficients, a greedy algorithm was made for each. The greed of the algorithms lies 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 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 finds the similarity, which is the same as in Alg. 4, since the potential number of intersections per hash function is doubled.

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

1.1.1 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 takes $\Theta((l-k+1)k)$ time.
line 6: `toMMSketch` for a string of length l, repeats H times $(l-k+1)$ comparisons. This takes $\Theta((l-k+1)H)$ time.
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 strings 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 runtime of $O(N^2 \cdot H)$.

Conclusively, the runtime will be $\Theta(N(l-k+1)(k+H)) + O(N^2 \cdot H)$. Assuming that $l = 1500$[1], it would necessitate an $N > 1500$ to conclusively say that the runtime is $O(N^2 \cdot H)$.

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

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

As the analysis is completely parallel to the one performed for **MM** immediately above, the runtime will be $\Theta(N(l-k+1)(k+H/2)) + O(N^2 \cdot (H/2))$, halving the number of hash functions needed.

Algorithm 4 Greedy Clustering using **MM**

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, \ldots, s_{N-1}\}$ 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 ← 0 to N − 1 do
4:     C[i] ← 0
5:     kT[i] ← kmerTransformation(s_i)
6:     MM[i] ← toMMSketch(kT[i])
7: c ← 0
8: for i ← 0 to N − 1 do
9:     if C[i] == 0 then
10:        c ← c + 1
11:        C[i] ← c
12:        for j ← 0 to N − 1 do
13:            intersections ← 0
14:            if C[i] == 0 then
15:                for k ← 0 to H − 1 do
16:                    if MM[i][k][0] == MM[j][k][0] then
17:                        intersections ← intersections + 1
18:                    else
19:                        if MM[i][k][1] == MM[j][k][1] then
20:                            intersections ← intersections + 1
21:                if intersections/H ≥ ε then
22:                    C[j] ← C[i]
```

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

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, \ldots, s_{N-1}\}$ 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 ← 0 to N − 1 do
    C[i] ← 0
    kT[i] ← kmerTransformation(s_i)
    MM[i] ← toMMhalfSketch(kT[i])
c ← 0
for i ← 0 to N − 1 do
    if C[i] == 0 then
        c ← c + 1
        C[i] ← c
        for j ← 0 to N − 1 do
            intersections ← 0
            if C[i] == 0 then
                for k ← 0 to H/2 − 1 do
                    if MM[i][k][0] == MM[j][k][0] then
                        intersections ← intersections + 1
                    if MM[i][k][1] == MM[j][k][1] then
                        intersections ← intersections + 1
                if intersections/H ≥ ε then
                    C[j] ← C[i]
```

1.2 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.

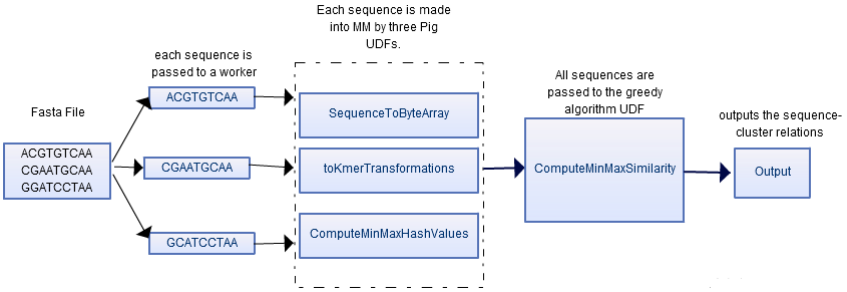

Figure 1: The flow of the MapReduce FrameWork using Pig

For the development of the MapReduce framework, Pig Latin was used. Pig Latin is a high level language for compiling and executing MapReduce jobs over Hadoop. Advantageous when performing a pipeline of MapReduce jobs [**?**], 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.

The Pig Script used to create this flow for the **MM** algorithm can be seen in Alg. 6. All functions that are not uppercase in the pig script are all user defined functions (UDFs) that were developed in the course of this paper. A closer examination of these UDFs will follow for comprehension of the work load behind it.

Before getting into the UDF, we should examine the datatypes of Pig Latin. The only variables that are allowed in a script are DataBags. These are unordered lists of data which contain all data gotten from an operation. As they are spillable, they register with a memory manager which allows for data of greater size than the available memory in a DataBag. These DataBags can then be parsed to a few SQL-like functions such as FILTER, FOREACH, LOAD or GROUP.

A noteworthy feature of these four functions is that they only run on a single element of the DataBag. For instance, LOAD first partitions the $INPUT file by newlines, and then parses it to the function right to USING. The same is the case for FOREACH, which takes each element of a databag and GENERATES and output using the Pig function or UDF to the rigth or GENERATE. The resulting output of each element is then gathered into a single DataBag, e.g. sequences in Alg. 6. Within the UDF an additional datatype is allowed, which

are Tuples. These are ordered lists of data, or virtually the array of Pig Latin. Tuples only exist within Pig UDFs, and are considered DataBags once they exit the UDF and go into the Pig Latin domain.

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

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, \ldots, s_{N-1}\}$ 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));
9: STORE results into $OUTPUT;

1.3 Pig Script analysis

Pig Scripts allow three languages to develop UDFs in: Python, Java, and javascript. Of these three, Java was chosen as it was the language with most features available for developing Pig UDFs. We shall examine each line to understand the purpose and thoughts behind each UDF.

In line 1, we are loading input from an $INPUT file using FastaReader. FastaReader was the only function not developed by the author, and is an open source input reader function[2] which can read fasta files so that each sequence is read as a single line. This means that if sequences span multiple lines, they are gathered by the FastaReader into a single line.

SequenceToByteArray then converts each sequence into a byte array using the same bitwise transformation as in the Additional Tools section, describing k-mer transformation. If the character N shows up, the sequence is returned as null since none of the test data that were used in this paper had any N present. The lines that hold an unknown character N are then filtered away in line 3.

In line 4 the k-mer for each sequence are made. The UDF toKmerTransformation was developed so that the output would be the same as using Alg. 1. This transformation was used as it was the optimal way of obtaining k-mers so that they took as little space as possible. In addition, using this implementation over `substring` is slightly quicker, as finding the transformed k-mer only requires a

[2] `http://cs.gmu.edu/~mlbio/MrMC-MinH/`

few multiplications in lieu of splitting strings.

For transforming the k-mers into **MM** sketches, the UDF ComputeMaxMinSimilarity was developed. It follows the exact same formula as Alg. 2, so that a Tuple is filled, in order, with a tuple of two elements containing the maximal and minimal hashed value of each hash function. The maximal and minimal hash value are both integers, as the hash functions are modulused with 10000 in the end, meaning that it would be pointless to use longs.

In line 6 and 7 two purposes are serves. Line 6 allowed to make a databag which in only one layer contained all tuples calculated in line 5. With this trick, it was rendered possible to let the FOREACH in line 8 only run once, as the `grouped` variable contains a DataBag with a single Tuple of all max- and minhashes of all sequences. Line 7 then uses line 6 to calculate the number of seuqences that have been transformed. By finding the number of sequences using line 7, and not simply calculating it in the UDF in line 8, it became possible to use MapReduce to find the number of sequences. This was faster than doing it in a single threaded UDF.

Finally, line 8 only runs once to fill the `results` variable. As the FOREACH only runs once, the advantage of MapReduce is eliminated, since only one worker is allowed to run ComputeMaxMinSimilarity. Unfortunately, there was no way of making it possible to compare each sequence to all other sequences in a MapReduce framework using Pig, since FOREACH cannot parse the entirety of a DataBag several times. Therefore, ComputeMaxMinSimilarity is only run once, and it runs the greedy algorithm as described in Alg. 4 line 7-22. It then returns a Tuple containing information on the cluster each sequence is located in. A notable difficulty that came into the creation of this algorithm was that in the beginning, the $\mathbf{MM}\frac{1}{2}$ algorithm ran much slower than the **MM**. After rigorous debugging, a hidden side effect of either the way Pig Latin handles Java UDF or JAVA itself turned out to cause this difference in speed.
In line 9, this Tuple is then stored into an $OUTPUT file.
The development of the $\mathbf{MM}\frac{1}{2}$ Pig script was almost identical to that of **MM**. The only differences is that the UDF used in line 8 was ComputeMaxMinHalfSimilarity instead. It follows the greedy algorithm described in 5 line 8-21. In the initial stages of development, i had written the Java version so that line 16-19 in Alg. 5 was written as

```
if(minhashArray[i][k] == minhashArray[j][k]){
        intersections++;
}
if(maxhashArray[i][k] == maxhashArray[j][k]){
        intersections++;
}
```

The corresponding line in ComputeMinMaxSimilarity for **MM** was

```
intersections = (minhashArray[i][k] == minhashArray[j][k] || maxhashArray[i][k]
```

Repeadedly, the runs of $\mathbf{MM}\frac{1}{2}$ were slower than those of **MM** for the same input, k and H, which theoretically made no sense. After a rigorous debugging, changing the above two if statements into the following two lines

```
intersections = (minhashArray[i][k] == minhashArray[j][k]) ? intersections + 1
intersections = (maxhashArray[i][k] == maxhashArray[j][k]) ? intersections + 1
```

increased the speed of $\mathbf{MM}\frac{1}{2}$ to realistically approximately double the speed of **MM**. This change in the implementation could not have been predicted to have such an influence on the runtime of the Pig scripts. The above was a perfect example of a hidden side effect of the language, for we see that the explicit if statements semantically are identical to the two '?' if statements, but still the '?' have a much faster runtime. While I may not have been able to determine all the hidden side effects in the implementation, as many were tried to be determined as possible.

Now that the considerations behind the algorithms, and their implementation have been describe, the testing of the implementation may begin.