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 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
- 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. **??**) was most applicable. These hash functions will be 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.

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
```

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 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 $l = 1500$[1], we would need a $N > 1500$ to conclusively say that the runtime is $O(N \cdot N \cdot H)$.

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

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 $\Theta(N(l-k+1)(k+H/2)) + O(N \cdot N \cdot (H/2))$.

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 ← 0 to N − 1 do
    C[i] ← 0
    kT[i] ← kmerTransformation(s_i)
    MM[i] ← toMMSketch(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 − 1 do
                    if MM[i][k][0] == MM[j][k][0] then
                        intersections ← intersections + 1
                    else
                        if MM[i][k][1] == MM[j][k][1] then
                            intersections ← intersections + 1
                if intersections/H ≥ ε then
                    C[j] ← C[i]
```

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

Input: a list of DNA/RNA sequences $S = \{s_0, s_1, ..., s_{N-1}\}$ 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.

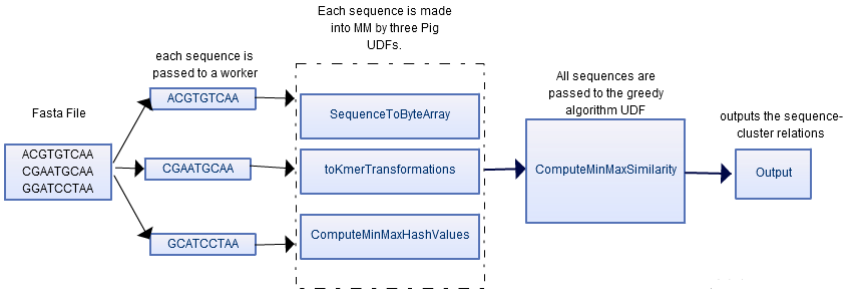

Figure 1: The flow of the MapReduce FrameWork using Pig

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.

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));
```