Project – Preprocessor for high throughput sequencing reads

Benjamin Argenson and Tim Wissel

October 29, 2017

1 Introduction

With the increased computational power and parallelisation techniques in modern computers, analyzation of massive amounts of DNA data has gained traction over the years. Novel fields and applications in biology and medicine are becoming a reality, beyond the genomic sequencing which was original development goal and application. For the analyzation of DNA, a fragmentation method is used to split up the DNA strings into short fragments. In result, millions of short DNA fragments can be sequenced in parallel. During sequencing, a so-called adapter sequence is used. This adds short fragments to the sequences. This results in a batch of sequences of which parts are not original DNA data. To analyze the sequences correctly, the adapter fragments should first be removed. In this project, our goals are to write an algorithm to do this preprocessing, and to predict the adapter sequence used.

2 Method

To solve the different tasks we mainly used suffix trees. To construct these trees we implemented the Ukkonen algorithm. For comparing the similarity between strings, we used Hamming distance.

2.1 Task 1

In task 1, an adapter sequence was given. To identify all the sequences that perfectly match a prefix of the adapter sequence, we developed the following methods:
Using the Ukkonen algorithm, we created a suffix tree that represents the suffixes of the reverse of the adapter sequence a. In result, this tree represented the reversed prefixes of a. Then, the structure of this tree was compared to all of the sequences to make suffix-prefix matches. To do this,

a depth-first traversal through the tree was made. When an internal node was reached, that node was pushed into the stack. Along the way, the terminal edges that were passed were kept in memory. Then, the edges of the tree of which the reversed labels were matched with the suffixes of the sequences.

```
1: function MatchingSuffix(adapterSequence, sequences)
2:     stack ← tree_root
3:     revAdapter ← adapterSequence                ▷ Reversed adapter sequence
4:     suffixTree ← Ukkonen(revAdapter)            ▷ Create suffix tree using
   Ukkonen's algorithm
5:     lengthDistribution ← Empty array
6:     numberOfSequences ← 0
7:     for S ∈ sequences do
8:         longestMatch ← matchLength(suffixTree, S)   ▷ The length of
   the longest match in the tree with S
9:         if length(longestMatch) > 0 then
10:            numberOfSequences ← numberOfSequences + 1
11:            if lengthDistribution_longestMatch > 0 then
12:                lengthDistribution_longestMatch ←
   lengthDistribution_longestMatch + 1
13:            else
14:                lengthDistribution_longestMatch ← 0
15:            end if
16:        end if
17:    end for
18:    return numberOfSequences, lengthDistribution
19: end function
```

2.2 Task 2

In task 2, some error margin was allowed. We approached this task using roughly the same methods as in task 1, but made some small changes.
When traversing the suffix tree, instead of looking for perfect suffix-prefix matches, suffixes and prefixes within a certain Hamming distance were also considered matches. This means that for every potential prefix-suffix match, the Hamming distance was calculated. Then, the hamming distance relative to the length of the strings was lower than some constant, the prefix-suffix combination was seen as a match.
This constant was set to require either 90% or 75% of the prefix and suffix words to be equivalent. The results for 90% and 75% similarity are shown in the Results section.

```
1: function MatchingSuffix(adapterSequence, sequences, mismatches)
2:     stack ← tree_root
3:     revAdapter ← adapterSequence                ▷ Reversed adapter sequence
4:     suffixTree ← Ukkonen(revAdapter)            ▷ Create suffix tree using
   Ukkonen's algorithm
5:     lengthDistribution ← Empty array
6:     numberOfSequences ← 0
7:     for S ∈ sequences do
8:         longestMatch ← matchLength(suffixTree, S, mismatches)    ▷
   The length of the longest match in the tree with S, with at maximum
   ¡mismatches¿ percentage of mismatches
9:         if length(longestMatch) > 0 then
10:            numberOfSequences ← numberOfSequences + 1
11:            if lengthDistribution_longestMatch > 0 then
12:                lengthDistribution_longestMatch                    ←
   lengthDistribution_longestMatch + 1
13:            else
14:                lengthDistribution_longestMatch ← 0
15:            end if
16:        end if
17:    end for
18:    return numberOfSequences, lengthDistribution
19: end function
```

2.3 Task 3

In task 3, we have to identify the adapter sequence that has been used in a set of sequences. For this, we also used suffix tree. For each sequences, we have created a suffix tree and for each suffix in each suffixes tree we have count the number of occurrences for this suffix. Then as we don't know the length of the adapter sequence, we decided to take the sequence of the same length that the sequences in the set that has the most common suffixes.

3 Results

3.1 Task 1

In task 1, we find that 646 364 sequences (over 1 million) contain suffixes that perfectly match a prefix of the adapter sequence (a). The practical running time of the algorithm is 2 minutes and 18 seconds.

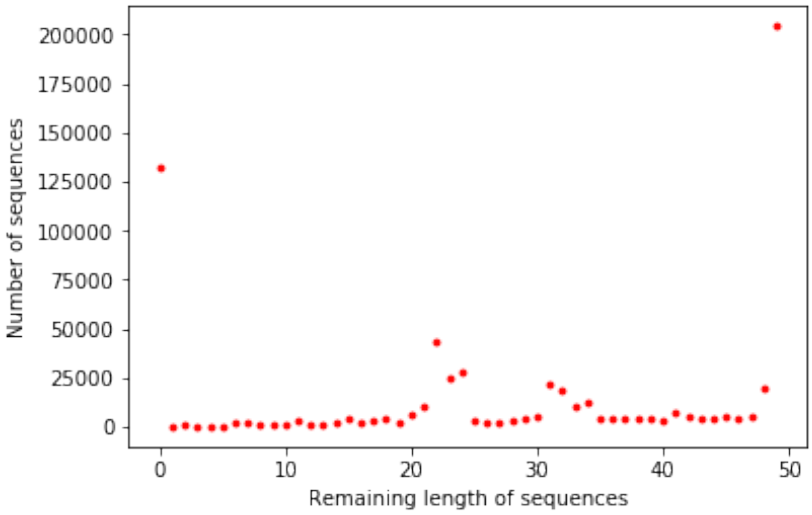

Figure 1: length distribution of the sequences that remain after we have removed the perfectly matching adapter fragments.

3.2 Task 2

Table 1 shows the amount of sequences that contain suffixes that match a prefix of the adapter sequence where this suffix can contain some percentage of mismatches.

Table 1: Task 2 results

Mismatch allowance	Running time (mm:ss)	Prefix-suffix matches
10%	7:15	670 918
25%	7:22	708 896

The following figures show the length of the sequences after removing the suffixes that matched a prefix of a.

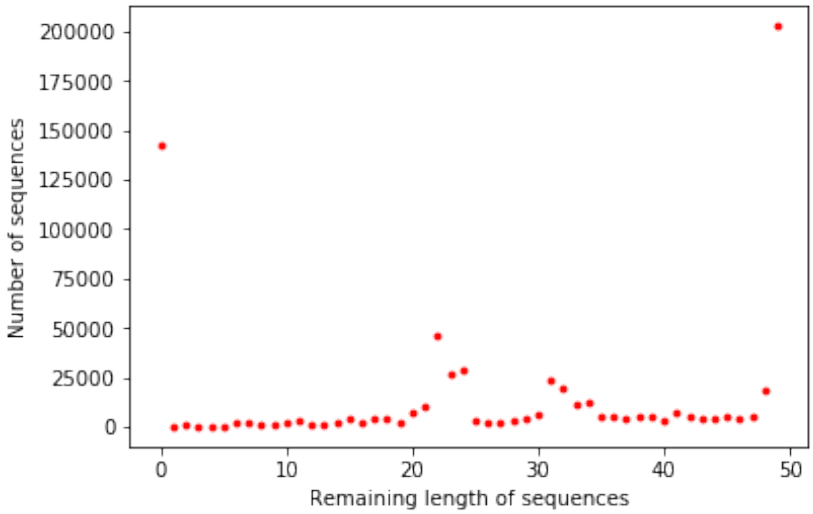

Figure 2: Length distribution of the sequences that remain after we have removed the imperfectly matching suffixes with up to 10% of mismatches.

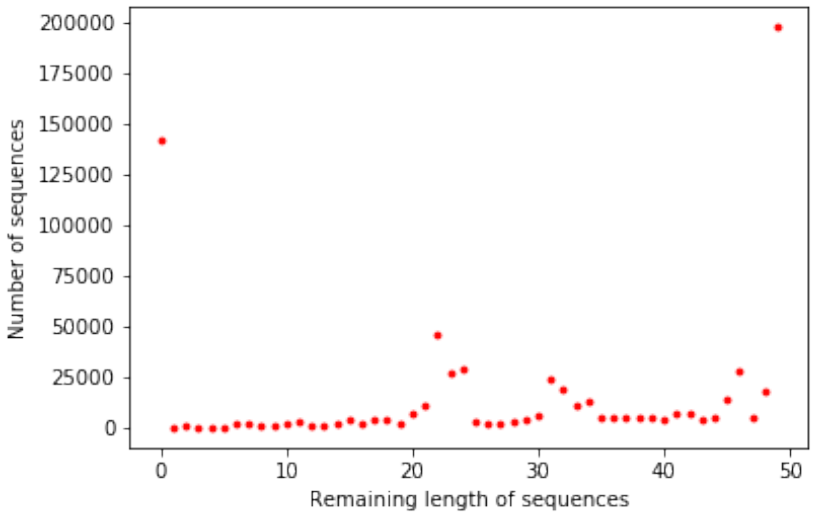

Figure 3: Length distribution of the sequences that remain after we have removed the imperfectly matching suffixes with up to 25% of mismatches.

3.3 Task 3

The most likely adapter sequence used for the sequences in the file 's_1-1_1M' is the following DNA sequence (with 362 216 occurences):

AGATCGGAAGAGCACACGTCTGAACTCCAGTCACATCA
CGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAA.

The length of the adapter that we should find being not indicated, we have taken the most likely adapter of the same length of the sequences.

The practical running time of the algorithm is 23 minutes and we can see the length distribution of the remaining sequence on the figure below (figure 4).
We also find an other common suffix pattern in this set which is very similar to the previous one since there is only one more 'A' at the end of the sequence.

For the file 'seqset3' our algorithm return the following sequence:
'AAGCTGCCAGTTGAAGAACTGTTGGAATTCTCGGGTGCCAAGGAACTCCAG'

And for the file 'S3-sequence' our algorithm return:
'TGGAATTCTCGGGTGCCAAGGAACTCCAGTCACACAGTGATCTCGTATGC'
(132 149 occurences over 1 000 000).

Moreover, the set in the last file contains additional common suffix pattern which can be represented by the following sequence:
'TTTCTATGATGAATCAAACTAGCTCACTATGACCGACAGTGAAAATACAT'
(84 387 occurences over 1 000 000)

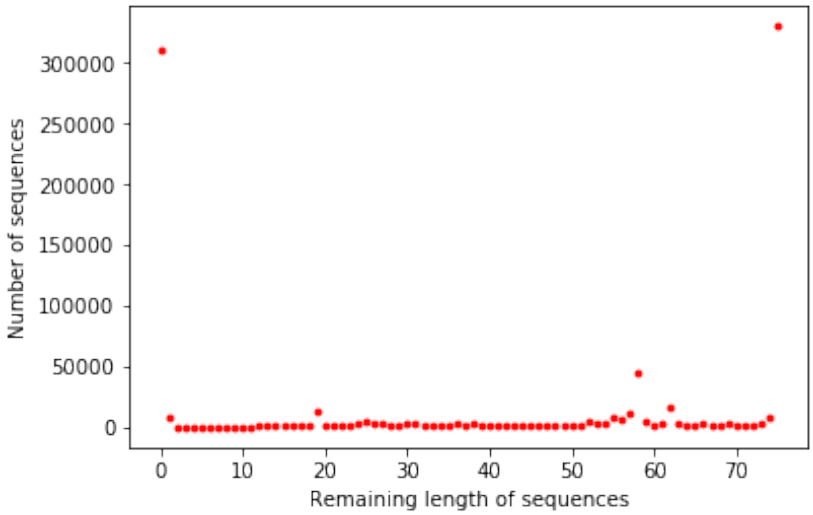

Figure 4: Length distribution of the sequences of the s1 file.

4 Discussion

4.1 Hamming distance

There are different methods to estimate the percentage of mismatches between suffixes and prefixes. For tasks 1 and 2 we decided to use Hamming distance. Other estimations like Edit/Euclidian/Jaccard distance could have been used.
Hamming distance measures the minimum number of substitutions that is necessary to transform one string into another. One issue with this is that this number can be relatively high when two strings look very similar. This can be the case when a string is shifted one or more positions, which is a quite common genetic copying fault. For example, the Hamming distance between AAGCTG and AAAGCT is 4, while the strings look quite similar. Edit distance may have been a better approach, because it takes into account deletion/insertions. This would have been a better way to tackle the problem where nucleotides are often added/lost in DNA fragment. In result, the amount of sequences found in task 2 is lower than expected.
The issue with Edit distance is that it takes a lot of time to compute. Its

runtime complexity is $O(d^2)$ where d is the length of the sequences (Andoni & Krauthgamer, 2006). The runtime complexity of computing Hamming distance is only $O(d)$ as only every character of both sequences needs to be read once. This is the reason we chose to use Hamming distance.

5 References

References

Andoni, A., & Krauthgamer, R. (2006). *The smoothed complexity of edit distance.*