Preprocessor for High Throughput Sequencing Reads

Emanuel Roos and Sophia Ohnemus

October 2019

Contents

1 Introduction

High throughput screening presents an important tool to lower the cost of DNA sequencing by producing millions of short DNA fragments and sequencing them in parallel. During this process specific DNA sequences, so-called adapter sequences, are added to beginning and end of the DNA fragments. Before being able to assemble the DNA fragments into one it is necessary to remove these adapter sequences from the DNA fragments. The goal of this project is to develop a preprocessor which is able to do so for perfectly and imperfectly matching adapter fragments or to determine the adapter sequence for a given dataset. Moreover, to reduce the sequencing costs even more sometimes multiple samples are sequenced at the same time and labeled with different adapter sequences (like a barcode), to be able to differentiate which sample the sequences are from. This preprocessor should also be able to work with such barcodes.

2 Methods

This section discusses the methods used for this project. First, section 2.1 shortly introduces the suffix tree data structure and shows two methods to build a suffix tree. Then, section 2.2 and section 2.3 deal with suffix-prefix matching of strings with and without allowing mismatches. Section 2.4 shows a method to find a possible adapter sequence for a set of sequences and section 2.5 deals with an algorithm to de-multiplex a barcoded library.

2.1 Suffix Tree

A suffix tree is a data structure that enables solving many different problems in string analysis efficiently. The tree contains all suffixes of a string T in its branches, but repeatedly occurring substrings are only represented once in the tree. Each edge is labeled with a substring of T and every inner node has at least two children, whose edge labels never start with the same letter. For every leaf node the labels on the edges on the path from the root denote the whole suffix of this branch. For multiple strings a generalized suffix tree can be build, which represents the suffixes of all strings, where each string ends with a unique terminal character. There are different methods to build a suffix tree and in the following, we will discuss a naive implementation as well as the so-called Ukkonen Algorithm [1].

Naive Construction

In the naive construction, every suffix T_{i+1} is added to the suffix tree representing all suffixes up to T_i by starting at the root and walking down the path whose label matches a prefix of T_{i+1} as far as possible. When the first mismatch is discovered a new node w is created before this point and a new edge out of w to a leaf node i that is labeled with the unmatched part of T_{i+1}. However, one can easily see that this implementation needs quadratic time [1].

Ukkonen Algorithm

The Ukkonen Algorithm is an algorithm to build a suffix tree in linear time. In each iteration i, for a given string $T\$$ the algorithm adds $T[i]$ to the suffix tree, so after the i-th iteration there is kind of a suffix tree for the prefix $T[..i]$. To do this in linear time, the costs for every iteration must be amortized $\mathcal{O}(1)$ and to achieve this, the algorithm makes use of different tricks [2].

- To save the costs for updating an existing edge in the tree in iteration i, the edges corresponding to the substring $T[b..i]$ are labeled with an index tuple $(b, \#)$, with $\#$ denoting the current end of the prefix which is set to the length of T in the end.
- After iteration i the variable *activepoint* is the longest suffix of $T[..i]$ which was already a substring of $T[..(i-1)]$. There are only new edges and leaves inserted in the tree if the *activeposition* requires this.
- The Algorithm uses $suffixlinks$ to get from a node s with $str(s) = ax$ and $a \in \Sigma$, $x \in \Sigma^+$ to the node w with $str(w) = x$ in constant time.
- By traversing the tree arbitrary long edges can be skipped in constant time.

2.2 Suffix-Prefix Matches

For two strings $S = s_1...s_n$ and $R = r_1...r_m$, we define a suffix-prefix match of S and R as the longest suffix of S that matches a prefix of R. To identify these suffix-prefix matches of a set of k strings with a given adapter sequence we can use the algorithm presented in the lecture. Here, first a generalized suffix tree containing all suffixes for each string as well as the adapter sequence is built and every leaf node S_j is labeled with a tuple j, n where j denotes the index of the string from which the suffix is from and n the start position of the suffix in the string. Moreover, each internal node v keeps a list $L(v)$ with the indexes j of its terminal edges, i.e. the edges which have only the string termination symbol as label, indicating a suffix ended in this internal node. Then the suffix tree is traversed depth-first and while doing that one keeps track of the terminal edges that were passed. Let p denote the index of the prefix sequence, then the procedure is as described in algorithm 1 [1].

Algorithm 1 Suffix-Prefix-Matches

```
use k stacks
traverse depth-first through the tree, v being the current node
    for each i in L(v)
        Push v on stack i
        if at leaf S_{p,0}
            record the path-label length of the node at top of stack i
    when backing up from v, pop all stacks i in L(v)
```

2.3 Suffix-Prefix Matches with Mismatches

To also take suffix-prefix matches into account where the suffix of S can contain up to a given percentage of mismatches to the prefix of R, again we first construct a generalized suffix tree but this time only for the set of suffix sequences. Then, we traverse all the branches of the tree and match the adapter as far as possible. However, if we discover a mismatch, we do not stop automatically but only if the mismatch counter is larger than a certain percentage of the adapter length. Then, at the end of each suffix branch that is reached we check again if the mismatch counter is also smaller than the mismatch rate when only considering the actual length of the suffix.

2.4 Finding the Longest Common Suffix

For a given set of sequences S we want to find the most likely adapter sequence, thus we want to find the longest prefix string R for which we can find the most suffix prefix matches with the set S. To do this we construct a generalized suffix tree for the set S. Then we traverse the tree depth first and save for every node how many terminal edges we passed on the way from the root to this node. In the end, we assume that the suffix corresponding to the branch with the most terminal edges on the path is the desired adapter string.

2.5 Finding Barcode Patterns

Here we assume that all the barcodes used have the same length and a minimum length *minimallength*. First, we remove the adapter sequence from the sequences by using the methods presented in section 2.2 and section 2.4. Then we construct a new suffix tree for the set of sequences removed from the adapter fragments. For every string we assume that the most likely barcode is the longest suffix that is shared by the most other strings. Finally, we norm all the barcodes that are found like that by the most occurring barcode length.

3 Results

For the following evaluations we implemented the algorithms described in section 2 in Python. To construct the suffix tree we first implemented the naive construction algorithm described in section 2.1. We planed to later switch to the linear time Ukkonen algorithm, but it turned out the gain would have been smaller than initially thought: the naive algorithm is only quadratic in the length of the strings, which is quite low at 50. Adding more strings on the other hand is possible in linear time. In other words, if m is the length of a single string and n the number of strings, the runtime is in $O(m \cdot n)$. Therefore, we stuck with the linear algorithm for this project.

Algorithm 2 Finding Barcode Patterns

```
1: procedure BARCODE(minimallength)
2:     Use 2 stacks, numberofsequences and lensuffix with length of amount of
3:     strings
4:     for node in suffixtree.leaves do
5:         if len(node.suffix) < minimallength then continue
6:         if len(node.stringIDs) > numberofsequences[stringID]
7:         or (len(node.stringIDs) = numberofsequences[stringID]
8:         and lensuffix[stringID] > lensuffix[stringID]) then
9:             for id in node.stringIDs do
10:            numberofsequences[id] ← len(node.stringIDs)
11:            lensuffix[id] ← len(node.wholesuffix)
12: Length of barcodes is most occuring length in lensuffix
13: Norm all other suffixes by this length
```

3.1 Perfectly Matching Adapter Fragments

The output data of a high throughput sequencing experiment with the adapter sequence a = "TGGAATTCTCGGGTGCCAAGGAACTCCAGTCACACAGTGATCTCGTATGCCGTCTTCTGCTTG" was given. We used the algorithm presented in section 2.2 to determine all sequences in the dataset that contain suffixes that perfectly match a prefix of a and at which position. In total, we found 646364 out of 1000000 sequences with a suffix-prefix match. The length distribution of the sequences that remain after removing the perfect matching adapter fragments can be seen in fig. 1. We can see that some sequences contain only the adapter sequence such that after removing it they become empty. On the other hand, some sequences do not contain the adapter sequence at all, so their final length stays at 50. Figure 2 shows that the algorithm used for this has a linear asymptotic runtime. The actual runtime of finding the suffix-prefix matches was only about 800 ms, after the construction of the suffix tree took about 30 minutes.

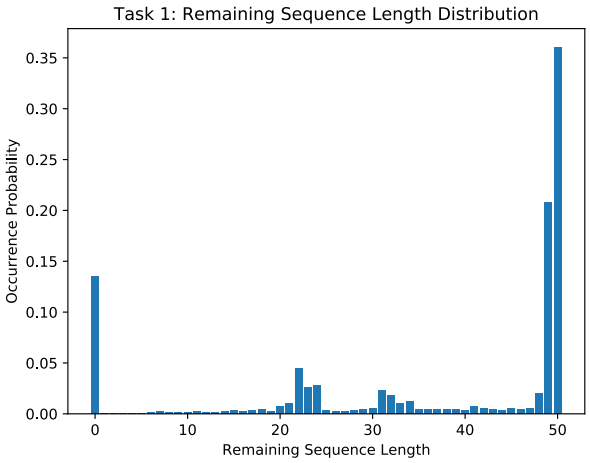


Figure 1: Length distribution of the sequences that remain after removing the perfectly matching adapter sequence

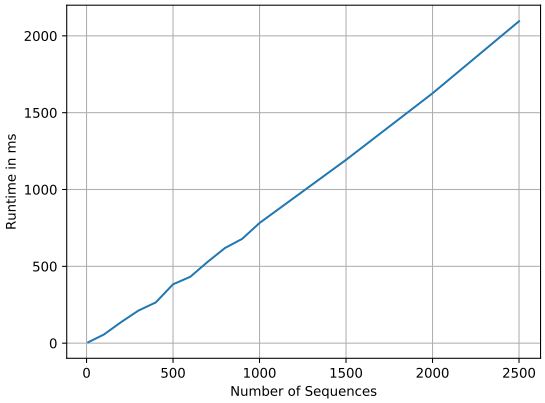
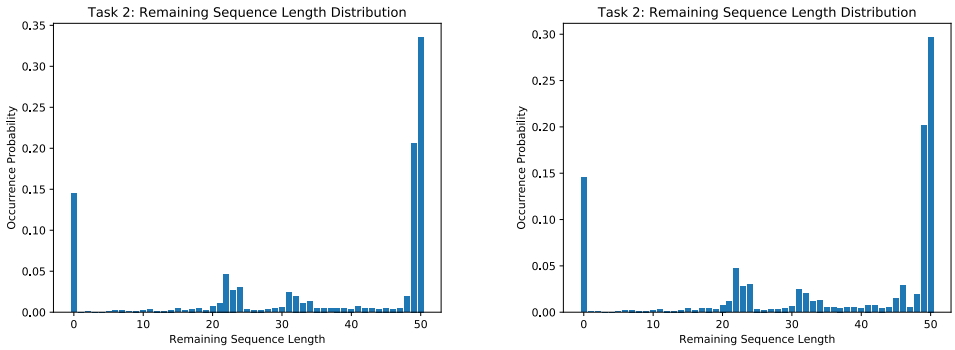


Figure 2: Runtime of algorithm for different numbers of sequences. The runtime of the used algorithm depends linearly on the number of sequences analyzed.

3.2 Imperfectly Matching Adapter Fragments

Next, we also took into account that sometimes sequencing errors can occur, causing the adapter sequence to not perfectly match the sequence suffixes. To find suffix-prefix matches with mismatches we used the algorithm described in section 2.3 on the dataset from 3.1. With a maximum mismatch rate of 10% there was a slight increase in matched sequences to 670918. For a mismatch rate of 25% this increased yet again to 708896. The length distribution of the sequences that remain after removing the imperfectly matching adapter fragments looks similar to the distribution for the perfectly matching

fragments (see fig. 3). The asymptotic runtime of this algorithm is approximately linear (see fig. 4). The actual runtime of finding suffix-prefix matches with mismatches was around 2 seconds for 10% mismatch rate and 7 seconds for 25% mismatch rate.



(a) Mismatch rate 10%

(b) Mismatch rate 25%

Figure 3: Length distribution of the sequences that remain after removing the adapter sequence with different mismatch rates

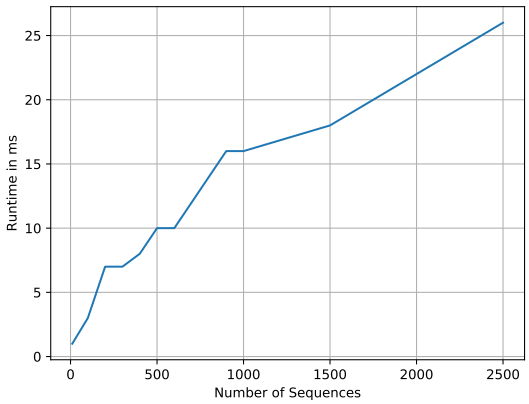


Figure 4: Runtime of algorithm for different numbers of sequences. The runtime of the used algorithm is approximately linear in the number of sequences.

3.3 Finding the Adapter Sequence

For our next dataset, the actual adapter sequence was unknown. For this reason, we applied the algorithm described in section 2.4 to find the most probable adapter sequence. Unfortunately, it turned out the runtime of our algorithm increased a little more than with the square of the number of sequences, meaning that it was too slow for the whole data set. Therefore, we had to do this task only with the first $100,000$ sequences of the

set, which took 8 minutes to construct the suffix tree and then 70 minutes to find the most common suffixes. The most likely adapter sequence we found through with our algorithm was

`CGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAAAGCACAAG`

Using this sequence and the perfectly matching algorithm explained in section 2.2 we removed the sequences from the adapter fragments. The length distribution of the sequences that remain can be seen in fig. 5. However, in fig. 6 we can see that the set also contains some highly frequent sequences that may bias the results of our algorithm. Moreover, the algorithm assumes that the sequence whose prefix matches the most suffixes is the adapter sequence, but still also the sequences that appear less often could be the adapter sequence. In table 1 we see that there are several sequences that have approximately the same frequency, so all of them are equally likely to be the adapter sequence. This indicates bias in the sequencing experiment.

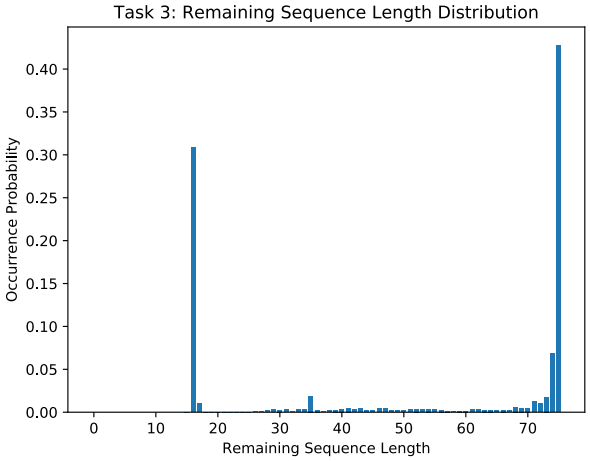


Figure 5: Length distribution of the sequences that remain after removing the adapter sequence.

Adapter Sequence	Frequency
CGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAAAGCACAAG	89769
CGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAAAGAAAAAAAAAACAGA	89768
CGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAAAGACCCAAAAAAAA	89768
CGTCTGAACTCCAGTCACATCACGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAAAAAGCCCCATATCAA	89768

Table 1: The four most likely adapter sequences.

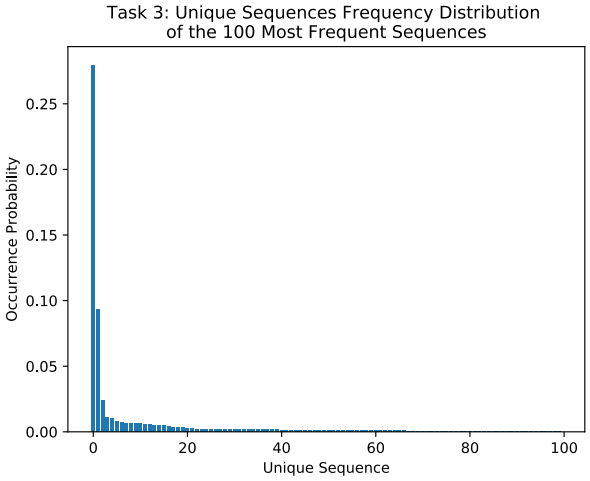


Figure 6: Frequency distribution of the 100 most frequent unique sequences

3.4 De-multiplex Barcode Library

Lastly, we analyzed a set of sequences that originate from different samples. The sequences are marked with a specific barcode to identify to which sample the sequence belongs to. We used the algorithm presented in section 2.5 to de-multiplex this barcode library. Unfortunately, the memory needed for the suffix tree was too much for our computers, so that we had to do this task with only the first $100,000$ sequences in the file. By applying the same procedure as in section 3.2 we found the sequence

`ATGCCGTCTTCTGCTTGAAACAA`

to be the most likely adapter sequence in the set. We used the procedure described in section 2.5 with $minimallength = 4$. The barcodes and the amount of sequences per sample can be seen in table 2. In total, 43 samples were analyzed in this set. The length distribution of the sequences per sample can be seen in fig. 7. We can see that the length of the sequences from the sample with barcode "AAAA" are nearly unaffected by removing the adapter sequence. Sequences of the sample with the "CGTA" barcode however are comparatively short. The most frequently occurring sequence per sample (if the sequence occurs more than one time) can be seen in table 3.

Barcode	Number of Sequences
AAAA	56856
CGTA	41542
GAAA	645
TGAA	284
TAAA	183
GGAA	103
TTGA	56
AGTA	35
TTAA	29
GTAA	27
CGGA	25
CTTA	20
AGAA	18
CAAA	14
TGGA	13
TGTA	12
CGAA	10
ACAA	10
GACA	9
GCAA	9
AACA	9
GGTA	7

Barcode	Number of Sequences
TTTA	6
CGCA	6
TGCA	6
CCCA	2
CATA	2
GGCA	2
CCAA	2
ATAA	2
TCAA	2
ANAA	2
GCCA	2
GTGA	1
AATA	1
ATGA	1
GNAA	1
CCTA	1
ACCA	1
AAGA	1
CTGA	1
AANA	1
TAGA	1

Table 2: Barcodes and number of sequences per sample.

Sequence with Barcode	Frequency
TAGCACCATCTGAAATCGGTTTATCATCGTATGCCGTCTTCTGCTTGAAAA	1219
TCTTTGGTTATCTAGCTGTATGATATCATCGTA	870
TCTTTGGTTATCTAGCTGTATGATATCATCGTATGCCGCCTTCTGCTTGAA	9
TGTAAACATCCTTGACTGGAAGCTTATCATCGTATGCCGCCTTCTGCTTGA	5
TCCCTGAGACCCTAACTTGTGATATCATCGGA	2
TTCAAGTAATCCAGGATAGGCTTATCATCGTATGCCGTCTTCTGCTTGTAA	2

Table 3: Most frequent sequences per sample

References

[1] G. Gausfield. *Algorithms on Strings, Trees and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[2] S. Rahmann. *Lecture notes in Algorithms on Sequences*. URL: http://ls11-www.cs.tu-dortmund.de/people/rahmann/algoseq/2016/03-1-index-suffixtree.pdf. 2016.

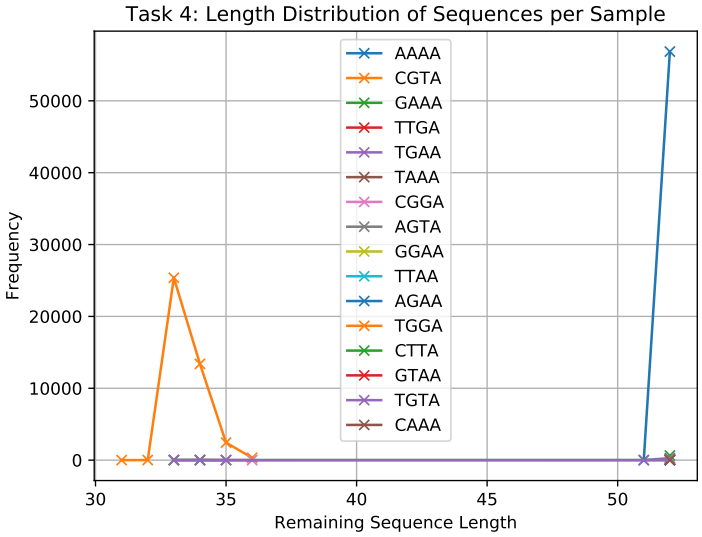


Figure 7: Sequence length distribution within each sample with more than 10 sequences per sample.