Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Alignment of Nanopore Sequencing Reads

Bachelor Thesis

2016
Rastislav Rabatin

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Alignment of Nanopore Sequencing Reads

Bachelor Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Computer Science
Supervisor:	Mgr. Tomáš Vinař,PhD.

Bratislava, 2016
Rastislav Rabatin

28804922

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Rastislav Rabatin
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Alignment of Nanopore Sequencing Reads
Zarovnávanie čítaní z nanopórového sekvenovania

Cieľ: Nanopórové čítania sú postupnosti elektrických signálov, ktoré možno približne preložiť do DNA sekvencií zhruba s 30% chybovosťou. Následne tieto približné DNA sekvencie zarovnávame k iným sekvenciám, no vysoká chybovosť zabraňuje použitiu efektívnych algoritmov alebo znižuje ich senzitivitu. Cieľom práce je navrhnúť nové prístupy ku zarovnávaniu nanopórových čítaní, ktoré obídu nespoľahlivý krok prekladania signálov do DNA sekvencií.

Vedúci:	Mgr. Tomáš Vinař, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	28.10.2015

Dátum schválenia: 29.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

28804922

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Rastislav Rabatin
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Alignment of Nanopore Sequencing Reads

Aim: Nanopore reads are sequences of electrical signals that can be approximately translated into DNA sequences with about 30% error rate. Usually these approximate DNA sequences are then aligned to other DNA sequences, however high error rates prevent the use of efficient and sensitive algorithms. The goal of the thesis is to design new approaches to nanopore sequence read alignment, bypassing unreliable translation of signals into DNA sequences.

Supervisor:	Mgr. Tomáš Vinař, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	28.10.2015
Approved:	29.10.2015

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgement

First, I would like to thank my supervisor Mgr. Tomáš Vinař,PhD. for his guidance and his patience mainly during the last days.

I am also thankful to doc. Mgr. Bronislava Brejová, PhD. and Mgr. Vladimír Boža for their helpful advice.

Special thanks to my family and friends for their support.

Abstrakt

Nanopórové sekvenovanie je DNA sekvenovacia technológia najnovšej generácie sekvenovacích technológií. Produkuje veľmi dlhé čítania, ktoré sú však náchylné na chyby. V tejto bakalárskej práci sa zaoberáme zarovnávaním týchto čítaní k referenčnej sekvencii.

Zarovnávanie nanopórových čítaní sa väčšinou robí v dvoch krokoch. Najprv sa rekonštruuje DNA sekvencia z dát, ktoré sú vyprodukované počas nanopórového sekvenovania, a potom sa použije nejaký bežný algoritmus na zarovnávanie sekvencií. Proces počas, ktorého sa rekonštruuje DNA sekvencia je náchylný na chyby, a teda počas toho sa stratí množstvo informácie. Navrhujeme prístup, ktorý sa snaží eliminovať túto stratu informácií a využiť ju počas zarovnania.

Kľúčové slová: skryté Markovove modely, nanopórové sekvenovanie, zarovnávanie čítaní

Abstract

Nanopore sequencing is a DNA sequencing technology of the most recent generation. It is known by its ability to produce long reads which are prone to errors. In this bachelor thesis we study the alignment of these high error rate reads.

The alignment of the nanopore reads is usually performed in two steps. First, we need to infer the DNA sequence from the data produced by the nanopore sequencing device and then we use a general-purpose alignment algorithm. However, the process of DNA sequence inference is prone to errors. Therefore much information is lost during this process. We propose an approach which tries to eliminate the loss of this information and tries to use it for the alignment.

Keywords: hidden Markov models, nanopore sequencing, alignment of reads

Contents

List of Figures

List of Tables

Introduction

Nanopore sequencing technology is a third generation DNA sequencing technology. It is based on measurements of electric current in the nanopore. When the DNA passes through the nanopore, it alters the electric current inside the nanopore. This change in current is measured and based on this measurements we can determine the sequence of DNA that passed through the nanopore. Unfortunately, this step of translation of electric current into DNA is very inaccurate.

The common task in bioinformatics is to align DNA sequences. There are many known algorithms that are used to solve this problem. Usually, heuristic algorithms are used because the exact algorithms are very slow.

The alignment of nanopore sequences is usually done in two steps. We translate the sequences of electric current levels into DNA sequences and then we use of the common algorithms to align these sequences between each other. This is quite inefficient because we loose a lot of information in the first step since the translation algorithm is inaccurate. Also the alignment algorithm is not exact and introduces additional error. We propose to merge these two tasks into a single step and use the information from the electric current measurements also in the alignment algorithm.

Our final goal is to align one nanopore read against a large database of reads. Our approach is to use a well-known paradigm, seed and extend (Section 4.5), to solve this problem. The usual approach for seeding this algorithm is to find the maximum exact matches. In our approach, we want to use a probabilistic model of the nanopore reads to find good seeds for our algorithm. This thesis is mainly focused on designing this probabilistic model, sampling DNA sequences from this model and finding the similar regions between the sampled sequences which can be used for seeding our alignment algorithm.

In the first chapter, we describe the nanopore sequencing technology in more detail. Second chapter describes hidden Markov models and the basic algorithms required in this thesis. In the third chapter, we describe the design of hidden Markov model for nanopore reads. Fourth chapter explains the problem of alignment of sequences and gives an overview of the algorithms that are used to solve this problem. We also explain the seed and extend paradigm and how to incorporate our hidden Markov model into this paradigm. Finally, in the last chapter we discuss the implementation details of our model and algorithms. We also discuss the experiments that we have done and analyze the results from these experiments. We mainly focus on analyzing the sampled sequences from our model which helps us to determine the good parameters for our heuristic algorithm for alignment of nanopore reads based on seed and extend.

Chapter 1

DNA Nanopore Sequencing

At first, we will give the brief introduction to bioinformatics and DNA sequencing. Then we will explain how nanopore sequencing technology works and compare this technology to other DNA sequencing technologies. We also describe the MinION sequencer from which we had data and what kind of data it provided us.

1.1 DNA

Most of the genetic instructions of living organism are stored in deoxyribonucleic acid (**DNA**). These instructions are used in many important processes of organisms such as protein synthesis. DNA is composed of two strands (**template** and **complement**) and has a shape of helix. Each strand is composed of molecules called nucleotides. There are four **nucleotides (bases)** – adenine, cytosine, thymine, guanine. Consequently, we represent each DNA strand as a sequence of letters A, C, T, G. When we know the sequence of one strand we can easily determine the sequence on the other strand by using the **base pairing rules**. Adenine (A) always pairs with thymine (T) and cytosine (C) always pairs with guanine (G) and vice versa.

1.2 DNA Sequencing

DNA sequencing is a process of determining the order of nucleotides in the DNA sequence from biological samples. There are many technologies that are used for sequencing DNA. Currently, we are working with the third generation of sequencing technologies.

One of the main problems with DNA sequencing is that no sequencing technology is able to read the whole sequence at once. Instead, sequencers produce many short subsequences of the DNA sequence called **reads**. The original sequence is then reconstructed by a heuristic algorithm combining individual reads into a longer sequence.

Sequencing technologies from the second generation were able to read only approximately 400 base pairs (bp) in a single read which is much shorter than a typical genome size. Human genome has approximately 3 Gb (gigabase pairs). With the new nanopore sequencing technology, we can sequence up to 100 kbp in a single read.

Number and length of the reads is not the only problem with sequencing. All sequencing technologies are prone to errors. Sequencer are not perfect. While sequencing technologies from the first and second generation achieved error rates less than two percent, nanopore sequencing technology suffers from very high error rates up to 30%.

1.3 Nanopore Sequencing Technology

In this section we describe the main idea of nanopore sequencing and basic terminology. In this thesis, we focused on the data from the MinION, produced by Oxford Nanopore Technologies (ONT).

The MinION device is a highly portable DNA sequencing instrument in the size of a regular USB flash drive. It is the first commercially sold nanopore sequencing device. As we mentioned above, the advantage of this device is that it produces very long reads compared to other sequencing technologies at a very low price. The disadvantage is that nanopore reads have very high error rate.

Nanopore is a small hole with an internal diameter of the order of one nanometer. Pores of this size are usually composed of proteins or graphene. Nanopores in MinION are so small that only a single strand of the DNA sequence can pass through it. MinION has multiple nanopores so it can read multiple sequences simultaneously.

As we can see in Figure 1.1, double-stranded DNA sequence is split into two strands which are connected by a hairpin adapter. Special adapters (proteins) are also attached at the beginning and at the end of the whole DNA sequence in order to lead the sequence through the nanopore. When we turn on the electric field DNA is attracted towards the anode which forces DNA sequence to travel through the nanopore. The template strand is driven through the nanopore first, followed by the complement strand. The speed at which DNA is going through the nanopore is controlled by the current and the adapters attached to the sequence.

When DNA sequence passes through the nanopore, it alters electric current flowing through the nanopore. **k-mer** is a sequence of k nucleotides which resides in the nanopore and influences the electric current flowing through the nanopore. Mathematical definition:

Definition 1. **k-mer** *is a word of length k over the alphabet $\Sigma = \{A, C, T, G\}$.*

The length of the k-mer is a given constant for the whole data set. We worked with a version of data from MinION in which $k = 5$ but there's a newer version in which $k = 6$.

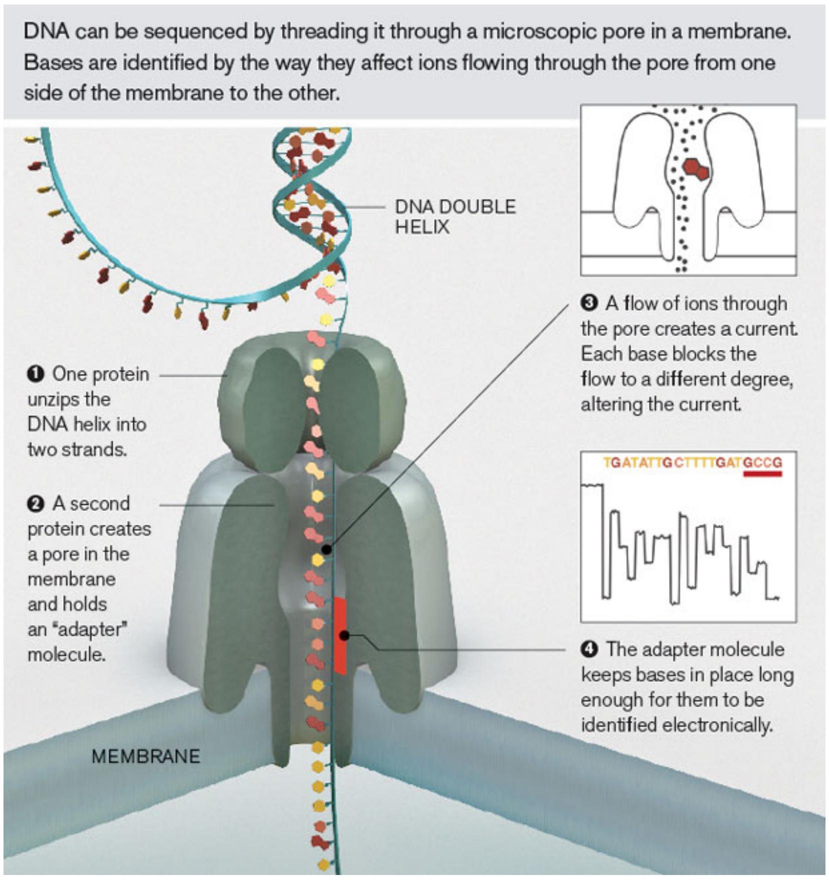
DNA can be sequenced by threading it through a microscopic pore in a membrane. Bases are identified by the way they affect ions flowing through the pore from one side of the membrane to the other.
DNA DOUBLE HELIX
❶ One protein unzips the DNA helix into two strands.
❷ A second protein creates a pore in the membrane and holds an "adapter" molecule.
❸ A flow of ions through the pore creates a current. Each base blocks the flow to a different degree, altering the current.
TGATATTGCTTTTGATGCCG
❹ The adapter molecule keeps bases in place long enough for them to be identified electronically.
MEMBRANE

Figure 1.1: Overview of the nanopore sequencing [Sch16]

The disruption in electric current is believed to be different for every k-mer. This change in current is measured thousand times per second. Raw electric current measurements are not usually stored in the output file because it would be very large. Instead, the MinION splits the sequence of measurements into sequence of **events**. **Event** is a short continuous sequence of electric current measurements, which in the perfect case, should correspond to a single k-mer.

You can image the sequencer as a head of a finite automaton that can read k letters at a time. The DNA strand that we want to read can be seen as a word on the input tape of this automaton. The head of the automaton moves in every step by a single letter to the right starting from the left. In the perfect case, every step of the finite automaton corresponds to a single event. For example, when the sequence is `ACTGCGT` and $k = 5$ then the first event should correspond to 5-mer `ACTGC`, the next event should correspond to `CTGCG` and the last event to `TGCGT`.

However, in the reality events can be split or merged because of an error in the event detection algorithm. We discuss these errors in Chapter 3.

The segmentation of measurements into events is done locally in the real time by the **MinKNOW** software running on the computer which the MinION is connected to by USB.

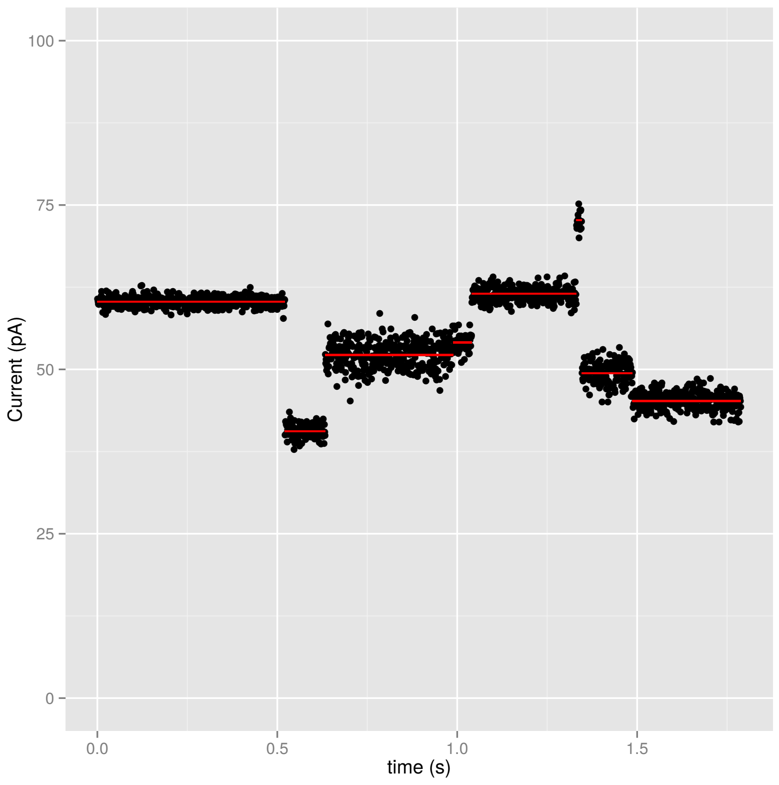

Figure 1.2: The segmentation of individual current measurements into events ([LQS15, supplement p. 2]). The black dots represent current measurements and the red line segments represent mean current levels measured during an event. All current samples that are in the interval marked by a red line segment are included in the event. Events can sometimes even overlap each other but usually they do not. We can also have gaps between the events.

Afterwards, these events are usually sent for **base calling** (translation of events to a sequence of letters A, C, T, G) to **Metrichor** (cloud-based computing platform). The events and the results of the base calling process are written to **FAST5** files. The MinION describes every event by four parameters: mean of current samples (in pA), standard deviation of the current (in pA), start of the event (in seconds) and length of the event (in seconds). More detailed description of FAST5 files and libraries for working with those files can be found in Chapter 5.

Metrichor performs basecalling for the template and complement strands separately. Sometimes it does not succeed and the FAST5 file contains only base call of the template strand. In the best case, the file contains both strands and additionally **2D base call**. 2D base call is constructed by creating consensus sequence from events from both strands by combining the information. Therefore the 2D base call is of a higher quality. The exact details of the base calling process are not provided by the manufacturer. We call the reads for which the 2D base call is available 2D reads. The reads for which only the template strand base call is available are called 1D reads.

1.4 Kmer probability model

Mean current level for every k-mer in the MinION is modeled by the Gaussian distribution. Parameters for this distributions are stored in the FAST5 file. Standard deviation of current for a given k-mer is modeled by the Inverse Gaussian distribution. In our work, we only use the mean current level. Therefore when we refer to the sequence of events, we only mean the sequence of the mean current levels corresponding to the events.

MinION provides parameters μ_x and σ_x for the Gaussian distribution for every k-mer separately. Since characteristics of the current measurements are slightly changing over time and are different for every nanopore, Metrichor uses scaling parameters in order to compensate. These scaling parameters are determined by Metrichor and saved into the FAST5 file together with the base called sequences.

The probability of observing an event with a mean current level e for a given k-mer x is:

$$P(e|x) = f(e, scale \cdot \mu_x + shift, \sigma_x \cdot var) \tag{1.1}$$

where μ_x and σ_x are given parameters for every k-mer x; $scale$, $shift$ and var are parameters given for every DNA strand, and

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.2}$$

We have used parameters computed by Metrichor in our model. We did not try to compute them by ourselves. Another possibility could be to replace these parameters by those computed by independently developed project, such as Nanocall [DDY+16].

Chapter 2

Hidden Markov Model

In this chapter, we describe hidden Markov model (HMM) which is a very common machine learning model used in many areas such as speech recognition, image processing and bioinformatics. We will explain the basic HMM terminology and classical algorithms used for this model which we also used in our work. In Chapter 3, we use this probabilistic model to define the probabilities of DNA sequences given a sequence of events from the MinION.

2.1 Definitions

Definition 2. Hidden Markov model with continuous emissions and silent states (HMM)
Let $H = (Q, S, E, q_0, e, t)$ where Q is a non-empty finite set of states, $S \subseteq Q$ is a set of silent states, E is an uncountably infinite set of emissions, $q_0 \in S$ is an initial state, $e : Q \times E \to [0, 1]$ is an emission probability function and $t : Q \times Q \to [0, 1]$ is a transition probability function. Then H is a hidden Markov model when the following conditions hold true:

1. $\forall u \in Q : t(u, q_0) = 0$
2. $(\forall u \in S)(\forall e \in E) : e(u, e) = 0$
3. $\forall u \in Q : \sum_{v \in Q} t(u, v) = 1$
4. $\forall u \in Q \setminus S : \int_{x \in E} e(u, x) dx = 1$

Notation 1. *We say that there is a transition from state u to state v, if and only if $t(u, v) > 0$. We will use notation $u \to v$ to denote this fact.*

2.2 Graph Representation

Hidden Markov models can be represented as directed graphs. In fact they are quite similar to the **finite automata** so they can be illustrated as state diagrams similarly to the finite automata. Transitions are directed edges (drawn as arrows) in the graph and states are vertices (drawn as circles). Initial state is drawn as two concentric circles. Transition probabilities are drawn above the edges. We will sometimes use graph terminology when talking about HMMs because graph representation is quite natural.

2.3 HMM as a Generative Model

HMM is a generative machine learning model and that is the reason why it is so similar to the finite automata. It can be viewed as an automaton which generates two sequences – sequence of emissions and sequence of states. The sequence of states is hidden. We can only see the sequence of emissions and that is the reason why emissions are sometimes called *observations*.

The automaton starts in the initial state and selects a random transition based on the transition probability function and moves through that edge to the next state. Afterwards, it generates a random element from the emission probability distribution of the current state and again moves to the next state based on the transition probability function. If the next state is silent then it does not emit anything and only moves to the next state. This process can go on forever but we are only interested in paths of a finite length.

Generative models directly define the joint probability over the pairs $P(\vec{\alpha} \wedge \vec{\beta})$ where $\vec{\alpha}$ is the input vector (observed – emitted vector) and $\vec{\beta}$ is output vector (vector that we want to predict). In case of HMM it is:

$$P(e \wedge s) = \left(\prod_{i=1}^{m} t(s_{i-1}, s_i) \right) \cdot \left(\prod_{i=1}^{n} e(v_i, e_i) \right) \tag{2.1}$$

where $e = e_1, e_2, \ldots, e_n$ is the sequence of emissions (input) and $s = s_0, s_1, s_2, \ldots, s_m$ is the sequence of hidden states (output) and $v_1, \ldots, v_n$ is a subsequence (not necessary continuous) of non-silent state of sequence s. For convenience, we define $s_0 := q_0$ (the initial state). We will use this convention throughout the whole thesis.

2.4 HMM and the Nanopore Sequencer

We can view the sequencer as a finite automaton which emits a sequence of events – the electric current levels and the internal (hidden) states are the k-mers of the sequence that is currently passing through the nanopore. We want to determine the sequence of k-mers that passed through the sequencer. This sequence will be represented by the path through the states in the model. Later (in the Chapter 3) we will explain why modelling the sequencer is not so straightforward and we also need to consider some internal characteristics of nanopores and associated errors during sequencing.

2.5 Viterbi algorithm

A common task when working with HMM is to find the most probable sequence of hidden states for a given emission sequence. More formally, we want to find

$$\operatorname{argmax}_s P(e \wedge s)$$

, where $s = s_0, s_1, \ldots, s_n$ is a sequence of states and $e = e_1, \ldots, e_m$ is a sequence of emissions. This can be done by using dynamic programming [DEKM98, p. 56].

We want to compute matrix V, where $V[i,u]$ is the probability of the most probable path ending in the state u and emitting the sequence $e_1, \ldots, e_i$. This matrix can be computed by using the following formulas:

$$V[0,u] = \begin{cases} 1, & u = q_0 \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

$$V[i,u] = \max_{v \in Q} V[i,v] \cdot t(v,u), \ \textbf{if } i > 0 \textbf{ and } u \in S \tag{2.3}$$

$$V[i,u] = \max_{v \in Q} V[i-1,v] \cdot t(v,u) \cdot e(u,e_i), \ \textbf{if } i > 0 \textbf{ and } u \in Q \setminus S \tag{2.4}$$

This matrix can be computed row by row starting from row zero and iterating through each row from left to right. When we have a transition going to a silent state then the value in the matrix V for the outgoing state has to be calculated before the value of the silent state for the same row of the matrix. In other words, the states have to be topologically ordered. Otherwise we would have a cycle in the recurrence. The final result is the $\max_{u \in Q} V[m,u]$.

To reconstruct the most probable path, we also need to store in every cell of the matrix the states for which we found the maximum value. Thus, every cell in the matrix stores a pair – the probability and a pointer to the state from which we computed the probability. Afterwards, the most probable path can be reconstructed by simply starting from the cell with the maximum value in the last row and following the pointers until we get into the first row of the matrix. To get the resulting path we need to reverse this path.

The time complexity of this algorithm is $\mathcal{O}(NMD)$ where N is the number of states, M is the length of the emission sequence and D is the maximum degree of a state in the HMM. The space complexity is $\mathcal{O}(NM)$.

2.6 The Forward Algorithm

The forward algorithm [DEKM98, p. 58] is very similar to Viterbi algorithm. It computes the matrix F, where $F[i,u]$ is the probability of ending up in the state u and emitting $e_1, \ldots, e_i$.

It can be computed by the following formulas:

$$F[0,u] = \begin{cases} 1, & u = q_0 \\ 0, & \text{otherwise} \end{cases} \tag{2.5}$$

$$F[i,u] = \sum_{v \in Q} F[i,v] \cdot t(v,u), \text{ \textbf{if} } i > 0 \text{ \textbf{and} } u \in S \tag{2.6}$$

$$F[i,u] = \sum_{v \in Q} F[i-1,v] \cdot t(v,u) \cdot e(u,e_i), \text{ \textbf{if} } i > 0 \text{ \textbf{and} } u \in Q \setminus S \tag{2.7}$$

Compare to the Viterbi algorithm, we have only replaced the maximum function with summation. The time complexity of this algorithm is the same as for Viterbi algorithm.

2.7 Sampling from the Conditional Probability Distribution

A very common usage of HMM is to find the most probable path in the model that explains the emission sequence using Viterbi algorithm (2.5). This algorithm only finds the optimal solution to the problem for a given set of parameters. The parameters highly influence the resulting most probable path. Sampling from HMM allows us to find many suboptimal solutions to the problem that still might be biologically significant [CP03]. We will also use this approach and sample from the conditional probability distribution defined by HMM to find many paths close to the optimal solution. This approach will allows us to explore the space of all the solutions in a randomized way. First, we will define the problem more formally and then describe the algorithm that we used for sampling.

We want to sample a random path $s = s_0, s_1, \ldots, s_n$ from the distribution which is defined by the probability mass function $f(s) = P(s|e)$ where $e = e_1, \ldots, e_m$ is a given emission sequence and $P(s|e)$ is the probability defined by the HMM.

Let P_u be the set of all paths starting in the initial state and ending in the state u. From the definition of the conditional probability we know:

$$P(s|e) = \frac{P(s \wedge e)}{P(e)} = \frac{P(e \wedge u)}{\sum_{q \in Q} \sum_{\pi \in P_q} P(e \wedge \pi)} = \frac{P(e \wedge u)}{\sum_{q \in Q} F[m,q]} \tag{2.8}$$

where $F[m, q]$ is the probability computed by the forward algorithm described in Section 2.6.

First, we will solve an easier problem and then show how we can solve the original problem using the algorithm for the easier problem. The easier problem is to pick a random path $u_0, \dots, u_n$ starting in the initial state and ending in a *particular* state u_n and emitting sequence of length m with the probability:

$$\frac{P(e \wedge u_0, \dots, u_n)}{F[m, u_n]} \tag{2.9}$$

Suppose that we have an algorithm that can solve this problem for every state sequence of length shorter or equal to n. If u_n is not a silent state then we can sample a random state u_{n-1} with the probability:

$$\frac{F[m-1, u_{n-1}]t(u_{n-1}, u_n)e(u_n)}{F[m, u_n]}. \tag{2.10}$$

Afterwards, we can use the hypothetical algorithm to sample a random path ending in the state u_{n-1} of length n and emitting sequence $e_1, \dots, e_{m-1}$. This way we got a random path of length $n+1$ ending in the state u_n with the probability:

$$\frac{F[m-1, u_{n-1}]t(u_{n-1}, u_n)e(u_n)}{F[m, u_n]} \cdot \frac{P(e_1, \dots, e_{m-1} \wedge u_0, \dots, u_{n-1})}{F[m-1, u_{n-1}]} = \frac{P(e \wedge u_0, \dots, u_n)}{F[m, u_n]} \tag{2.11}$$

Which is the result that we wanted to achieve.

If u_n is a silent state then we can again sample a random state u_{n-1} with the probability:

$$\frac{F[m, u_{n-1}]t(u_{n-1}, u_n)}{F[m, u_n]}, \tag{2.12}$$

and then use the algorithm for sampling a random path ending in the state u_{n-1} of length n and emitting sequence $e_1, \dots, e_m$. This way we obtain a random path of length $n+1$ ending in the state u_n with the probability:

$$\frac{F[m, u_{n-1}]t(u_{n-1}, u_n)}{F[m, u_n]} \cdot \frac{P(e_1, \dots, e_m \wedge u_0, \dots, u_{n-1})}{F[m, u_{n-1}]} = \frac{P(e \wedge u_0, \dots, u_n)}{F[m, u_n]} \tag{2.13}$$

Now, we can solve the original problem using this algorithm. We sample the last state u_n with the probability:

$$\frac{\sum_{\pi \in P_{u_n}} P(e \wedge \pi)}{\sum_{q \in Q} \sum_{\pi \in P_q} P(e \wedge \pi)} = \frac{F[m, u_n]}{\sum_{q \in Q} F[m, q]}, \tag{2.14}$$

and then we use the algorithm mentioned above the sample a random path ending in u_n. The probability of the whole path obtained this way is:

$$\frac{F[m, u_n]}{\sum_{q \in Q} F[m, q]} \frac{P(e \wedge u_0, \dots, u_n)}{F[m, u_n]} = \frac{P(e \wedge u_0, \dots, u_n)}{\sum_{q \in Q} F[m, q]}$$

which is exactly Equation (2.8).

In summary, the algorithm starts with computing the matrix F by the forward algorithm (Section 2.6) and then it samples the random path starting from the end (the last row of the matrix). Then it picks a random state according to the probability of in the last row of matrix F. The probabilities are normalized by the sum of the row. Then we pick a random edge going to the chosen state according to the sum of probabilities of all paths going through each edge and ending in the chosen state. We repeat this process until we get to the initial state.

The time complexity of the preprocessing phase (the forward algorithm) is $\mathcal{O}(NMD)$ where N is the number of states, M is the length of the emission sequence and D is the maximum degree of a state in the HMM. The space complexity of this phase is $\mathcal{O}(NM)$. The time complexity of the sampling phase is $\mathcal{O}(kLD)$, and space complexity is $\mathcal{O}(kL)$, where k is the number of samples, and L is the length of the sampled sequence. If there are no silent states then $L = M + 1$, otherwise L can even reach NM (we can visit all the cells in the matrix, but we cannot return to any cell because the silent states are ordered in the topological order as we mentioned in the Section 2.5). The size of L highly depends on the specific architecture of the HMM.

We can also trade space for time to speed up the sampling phase. The preprocessing phase will be only slowed down by a constant. The goal is to improve the sampling phase. In addition to storing the original probabilities from the forward algorithm, we can also store the individual summands that were added to the sum which is stored in a particular cell. Then we can normalize them by the total sum and compute prefix sums which causes the increase of the space complexity to $\mathcal{O}(NMD)$. However, this additional space allows us to omit the computation of the summands and we can just generate a random number x in the interval $[0, 1]$, and find a number in the prefix sums array that is greater or equal than x. Since the sequence of prefix sums is non-decreasing, we can use binary search which decreases the time complexity of the sampling phase to $\mathcal{O}(kL \log(D))$.

Chapter 3

HMM for Modelling MinION Data

We will start with a description of the simplest model for our data assuming no event detection errors. Then we will describe event detection errors and problems with nanopore sequencing which we need to deal with and incrementally show how we can modify our HMM to account for these issues. We will describe our model for $k = 5$, which corresponds to an older version of MinION. However, the approach can be easily generalized to larger values of k (current MinION chemistry uses $k = 6$).

3.1 Model Without Errors

In the simplest HMM, we assume that the sequencer moves stepwise from 5-mer to 5-mer, and emits a single event from each 5-mer. The architecture of this HMM is quite similar to de Bruijn graphs [Wik16a]. De Bruijn graphs are used to represent overlaps between reads during reconstruction of DNA sequences from many overlapping reads [ZB08]. In our model, we will have states corresponding to 5-mers, and edges representing exact overlaps of length four.

For convenience we also define these mappings:

Definition 3. *We define mappings* $\mathrm{move}_i : \Sigma^5 \to 2^{\Sigma^5}$ *as:*

$$\forall x_1, x_2, x_3, x_4, x_5 \in \Sigma, i \in \mathbb{N} : \mathrm{move}_i(x_1x_2x_3x_4x_5) = \{x_{i+1} \dots x_5 y | y \in \Sigma^i\}$$

Sometimes we will use $\mathrm{move}(x)$ *to denote* $\mathrm{move}_1(x)$.

Mapping $\mathrm{move}_i(x)$ basically maps 5-mer x to a set of 5-mers which the sequencer might emit after i steps assuming no errors. If we take errors into account then the set of possible 5-mers would be greater. A different way of looking at this mapping is that it left-shifts the sequence by i symbols and adds arbitrary symbols from Σ at the end of the sequence to get a sequence of length five.

For example:

$$\mathrm{move}_0(ACTGC) = \{ACTGC\}$$

$$\text{move}_1(ACTGC) = \{CTGCA, CTGCC, CTGCT, CTGCG\}$$

$$\begin{aligned}\text{move}_2(ACTGC) = \{&TGCAA, TGCAC, TGCAT, TGCAG, TGCCA, TGCCC,\\ &TGCCT, TGCCG, TGCTA, TGCTC, TGCTT, TGCTG,\\ &TGCGA, TGCGC, TGCGT, TGCGG\}\end{aligned}$$

$$\vdots$$

$$\forall k \geq 5 : \text{move}_k(ACTGC) = \Sigma^5$$

Our HMM will have states $Q = \Sigma^5 \cup \{0\}$, where 0 is the initial state. The set of all emissions is $E = \mathbb{R}$.

Every state in our HMM has four outgoing edges. Every edge represents one step of the sequencer. For every state x we have directed edges to all states $y \in \text{move}(x)$. We also have an initial state. In summary, we have $N = 4^5 + 1 = 1025$ vertices and $M = 4^5 \cdot 4 + 4^5 = 5120$ edges.

Emission probabilities are computed from the probability density function of Gaussian distribution and all the parameters for this distribution are taken from Metrichor FAST5 files as described in Section 1.4 by using Equation (1.1).

Transition probabilities for the initial state come from the uniform distribution i.e.:

$$\forall x \in \Sigma^5 : t(0, x) = \frac{1}{4^5} \tag{3.1}$$

Transition probabilities for other states also come from the uniform distribution i.e.:

$$\forall x, y \in \Sigma^5 : t(x, y) = \frac{1}{4} \tag{3.2}$$

3.2 Event Detection Errors

Our simple HMM assumed the perfect case when no event detection errors occur. Unfortunately, we may encounter two types of errors [LQS15, supplement p. 3]:

1. **Skipped event.** Event might be skipped by the event detection algorithm if the current level measured for the previous event was very similar to the current level for the next event.

2. **Split event.** The same event may be emitted multiple times because of transient noise that might look like change in current. Event detection is done in real time which makes it harder to detect the right intervals for events.

Probably the biggest problem arises with long sequences composed of the same nucleotide called **homopolymers**. These kind of sequences can occur in DNA quite often. For example poly-A tail is a long sequence of adenine (A). In case of homopolymers, the event detection algorithm only sees a very long sequence of measurements where the current is very similar because the measurements come from the same distribution. It is very hard to segment these kind of sequences into events in real time. Therefore we never really know, how long are these sequences.

3.3 Modelling Event Detection Errors in HMM

3.3.1 Split Events

Duplicated events can be modelled very easily. We will just add transitions from every state to itself (loops). This way, we also allow HMM to generate homopolymers.

3.3.2 Skipped events

First, we want to model only skips of one event by the sequencer – skips of a single base.

Definition 4. *We will use the terminology* **skips of length** k *to denote the fact that the sequencer skipped k events (bases) in a row. More formally, transition $u \to v$ is a skip of length k for a state u if and only if $v \in \mathrm{move}_{k+1}(u)$ and skips of length k are all transitions in*

$$\{u \to v | u, v \in Q \land v \in \mathrm{move}_{k+1}(u)\}$$

Definition 5. *When we have a transition from state u to state $v \in move_i(u)$ then we call it a* **move of length** i.

Consider state x. Normally, we can transition to any $y \in \mathrm{move}_1(x) \cup \mathrm{move}_0(x)$. In order to deal with skips of length one, we add transitions to states $\mathrm{move}_2(x)$. That is $4^2 = 16$ new edges for each state.

If we want to model skips of length two in this way, we need to add additional $4^2 + 4^3 = 80$ edges for every state compared to the simplest HMM. That is also edges to states $\mathrm{move}_3(x)$ for every x. If we continue in this fashion, we need to add $4^2 + 4^3 + 4^4 = 336$ edges for skips of length three, and so on. In general, if we model skips of length k then we have

$$\sum_{i=0}^{k+1} |\,\mathrm{move}_i(x)| = \sum_{i=0}^{k+1} 4^k = \frac{4^{k+2} - 1}{3} \tag{3.3}$$

transitions for every state. This sum includes loops (mentioned in the previous section 3.3.1), transitions from the simplest HMM (section section 3.1), and skips. Note that the number of edges grows exponentially and we can get a complete graph very quickly. Working with complete graph is very computationally expensive. It makes all the algorithms that we will need very slow and requires a lot of memory.

Some of the skips might lead to the same state. For example, consider state `ACTTT`; then one of the skips of length one goes to `TTTCG`, but we could also say that transition from `ACTTT` to `TTTCG` is a skip of length two. In other words, when we have state $x = x_1x_2zzz$ then we have skip of length one going to $zzzab$ for some $a, b \in \Sigma$ and skip of length two also going to $zzzab$ because $zzzab \in \text{move}_2(x) \wedge zzzab \in \text{move}_3(x)$. We do not need to have multiple edges going to the same state. We just merge those edges into one edge and sum up all transition probabilities in order to preserve the property that sum of all the transition probabilities going from one state is one. If we allow skips of an arbitrary length then we have $\mathcal{O}(N^2)$ edges: two edges between every node going opposite directions and loops.

Notice that when we have 5-mer composed of the same bases (e.g. `AAAAA`) then loop goes to this state, normal edge goes to this state, and also all the skips of an arbitrary length k go to this state which causes the problem with determining the exact length of homopolymers.

3.3.3 Transition Probabilities

The transition probabilities can be learnt from the data. The classical approach is to set the probability of transitioning from the state u to the state v as:

$$t(u, v) = \frac{c_{u,v}}{\sum_{w \in Neigh_u} c_{u,w}} \tag{3.4}$$

where $c_{u,v}$ is the number of transitions from u to v that we have observed in the training data set plus pseudocount p and $Neigh_u$ is the set of all neighbours of the vertex u. We add pseudocount p to $c_{u,v}$ to ensure that we do not get zero probability. The data analysis and a more detailed description of the training phase can be found in Chapter 5.

3.4 Profile HMMs

Profile hidden Markov models are used for profile analysis of related sequences [DEKM98, chap. 5]. Profile is a description of an alignment of multiple sequences, for example profile of a protein family. Proteins in the same family have similar functions and are evolutionary-related. When we have HMM for a given family, we can generate protein sequences which are very similar to the proteins in the family or we can test if the given sequence is a member of the family. Profile HMM has three types of states:

1. **Match** - corresponds to a substitution or a match in the alignment.

2. **Delete** - corresponds to a deletion in the alignment. This state is *silent* which means that it emits nothing. All the other states have emissions.

3. **Insert** - corresponds to insertions in the alignment.

Every position in the alignment has these three types of states. If the length of the alignment is L then we have $3L$ states. See the typical profile HMM in Figure 3.1.

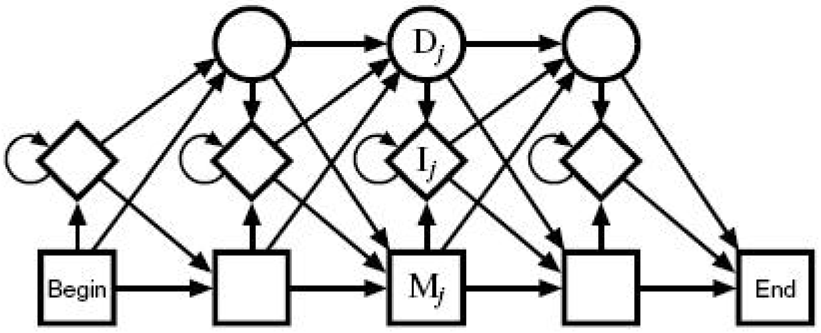

Figure 3.1: Typical profile HMM [DEKM98]. State denoted by M_j is the match state, D_j is the delete state and I_j is the insert state. This HMM has also has a special state for the end of the sequence which is a silent silent.

3.4.1 Adaption of the Profile HMM to Our Problem

We can think of errors in the sequence as insertions and deletions. Therefore we can adapt profile HMM to solve our problem.

Every 5-mer has three states – match, delete and insert. We denote these states for 5-mer x by M_x, D_x and I_x, respectively. In our problem we define these states as:

1. **Match(M_x)** - represents emission of event from the distribution of 5-mer.

2. **Delete(D_x)** - represents a skipped event corresponding to 5-mer x. When we add transitions between delete states then we basically allow skips of an arbitrary length without having N^2 edges, which solves the problem with the previous model.

3. **Insert(I_x)** - represents a *split event* for 5-mer x or insertion of an unexpected event after x corresponding to 5-mer which is not in move(x). By looking at the data, we can notice that split events occur quite often and insertions of unexpected events are rare. We did not model unexpected events in the previous design of our HMM in Section 3.3.

We can also adapt transitions from the typical profile HMM which you can see in Figure 3.1. Every node for 5-mer x (M_x,I_x or D_x) has transitions to all states in this set:

$$\{I_x\} \cup \{M_y | y \in \text{move}(x)\} \cup \{D_y | y \in \text{move}(x)\}$$

Let us look at the common paths in the model. The path for a read without errors uses only match states.

The path modeling skipped event for 5-mer y starts in M_x, D_x or I_x such that $y \in \mathrm{move}(x)$ and then it goes to D_y. Afterwards, it can transition to a next delete or match state for an arbitrary $z \in \mathrm{move}(y)$.

The path modelling a split event for 5-mer x starts in M_x and then goes to I_x and loops there couple times and then continues with a transition to the next delete or match state. It does not really make sense to have transition $D_x \to I_x$ in our case so we will not include them in our model. Transitions from I_x to D_y for $y \in \mathrm{move}(x)$ seem to be quite rare, but they make sense. They correspond to a duplication of an event followed by a skipped event in the sequence.

Note that we still have the same problem with homopolymers. Since duplicated events are common errors then we cannot differentiate between 5-mer `AAAAA` as being duplicate (transition to state I_{AAAAA}) or another correct event (transition to state M_{AAAAA}). The current for a homopolymer should be approximately the same all the time.

Chapter 4

Alignment of Sequences

In this chapter, we first explain what is alignment of sequences and why it is important in bioinformatics. Then we describe basic algorithms that are used for solving this problem.

4.1 Motivation

Probably the most common and crucial task in bioinformatics is to compare two DNA sequences. The purpose of aligning two sequences (pairwise alignment) is to identify similar regions. For example, this can help to identify regions which changed over the course of evolution or identify mutations in DNA of a virus. When we find similar regions in many sequences of different organisms it might mean that the region encodes information that is important for the organisms.

4.2 Problem Statement

Definition 6. Alignment of two sequences (pairwise alignment)
Consider two sequences $u = u_1 \dots u_n$, $v = v_1 \dots v_m$ and a matrix

$$M = \begin{pmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,k} \\ M_{2,1} & M_{2,2} & \dots & M_{2,k} \end{pmatrix}$$

. We call the matrix M an alignment of sequences u and v (pairwise alignment) if the following conditions hold true:

1. *All the elements of the matrix are symbols over the alphabet $\{A, C, T, G, -\}$*

2. *$M_{1,1}M_{1,2} \dots M_{1,k}$ is a word created from u by insertion of dashes.*

3. *$M_{2,1}M_{2,2} \dots M_{2,k}$ is a word created from v by insertion of dashes.*

4. *We don't have two dashes in the same column.*

For example, let $u = ACTGAAAATA$ and $v = ACAGGATTA$. Then one possible alignment of u and v is:

```
ACTGAAA--ATA
AC---AGGATTA
```

There are four types of columns in the pairwise alignment. Every type of column has its name. A column with two same letters is called match, a column with two different letters is called mismatch, dash above a letter is called insertion and a letter above dash is called deletion.

There are many valid alignments for two sequences. For example, the following alignment is also a valid for the sequences mentioned above, albeit it is not very useful:

```
ACTGAAAATA---------
----------ACAGGATTA
```

In most cases, our goal is to find the "optimal" alignment is some **scoring system**. The scoring system usually assigns a score to every column of the alignment and the final score for the alignment is the sum of scores for all of the columns. For example, we may have a scoring system which assigns score $+1$ to every match column and -1 to every mismatch, insertion and deletion. The optimal alignment will be the one with the greatest score.

When we are constructing the scoring system, we are not limited to setting the score only for the types of columns mentioned above. For example, we can also say that we want to give a higher score to columns with letters A and C.

The choice of the scoring system can highly influence the resulting optimal alignment. Usually, the goal is to design a scoring system that reflects the evolutionary history of the sequences.

In general, when we have some scoring system we define two types of problems.

Definition 7. Global alignment problem
Input to the problem are two sequences x and y. The output is a pairwise alignment of x and y with the best score according to the scoring system.

Definition 8. Local alignment problem
Input to the problem are two sequences $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$. The output is a pairwise alignment of two sequences $x_i \dots x_j$ and $y_k \dots y_l$ with the best score according to the scoring system for some $1 \leq i \leq j \leq n$ and $1 \leq k \leq l \leq m$.

4.3 Needleman–Wunsch Algorithm

Needleman–Wunsch algorithm [DEKM98, p. 19] is used for solving the global alignment problem. Let us demonstrate the algorithm using a simple scoring system: $+1$ for match and -1 for mismatch, deletion or insertion. The goal is to find the alignment with the largest score. The idea of this algorithm is based on dynamic programming.

Consider input sequences $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$. The algorithm computes matrix A, where $A[i, j]$ is the largest score for alignment of sequences $x_1 \dots x_i$ and $y_1 \dots y_j$.

Let us start with the trivial case when $i = 0$ or $j = 0$. For $i = 0$ we are aligning empty sequence to the sequence $y_1 \dots y_j$. There is only one way to do it: write dash above each letter in $y_1 \dots y_j$ which corresponds to an insertion. The score of this alignment is $-j$. Similarly, in case of $j = 0$ we get score $-i$.

Now, consider the case when $i, j > 0$. We will define a function $s(a, b)$ as:

$$s(a,b) = \begin{cases} 1, & a = b \\ -1, & \text{otherwise} \end{cases} \tag{4.1}$$

Last column of the alignment for the sequences $x_1 \dots x_i$ and $y_1 \dots y_j$ must conform to one of the possibilities.

First possibility is that the last column may contain x_i and y_j. Then the score for that column is $s(x_i, y_j)$. If we remove the last column of the alignment, we get the alignment for sequences $x_1 \dots x_{i-1}$ and $y_1 \dots y_{j-1}$. We claim that this alignment must be the optimal alignment of sequences $x_1 \dots x_{i-1}$ and $y_1 \dots y_{j-1}$. To prove this claim, consider that there could exist an alignment with a higher score. Then by removing the last column with x_i and y_j we would end up with an alignment of sequences $x_1 \dots x_i$ and $y_1 \dots y_j$ with a higher score than the original alignment which contradicts the fact that we took the optimal alignment. Therefore the total score for the best alignment with the assumption that the last column contains x_i and y_j is $A[i-1, j-1] + s(x_i, y_j)$.

The second possibility is that the last column contains x_i and dash. Then the score for this column is -1 and we can again remove this column to get the alignment of sequences $x_1 \dots x_{i-1}$ and $y_1 \dots y_j$. The best score for this alignment is $A[i-1, j]$. Therefore the total score for the original alignment is $A[i-1, j] - 1$.

Finally, the third possibility is that the last column contains y_j and dash. Then the score for this alignment is $A[i, j-1] - 1$.

Therefore, $A[i, j]$ can be computed as:

$$A[i,j] = \max \begin{cases} A[i-1, j-1] + s(x_i, y_j) \\ A[i-1, j] - 1 \\ A[i, j-1] - 1 \end{cases} \tag{4.2}$$

Every cell in the matrix can be computed from the three adjacent cells: the cell above, the cell on the left-hand side and the top-left cell. Therefore, we can iterate over the matrix row by row from the left-hand side and calculate values of every single cell in the matrix. At the end of this computation, the final result will be located in the cell $A[n, m]$.

The time and space complexity of this algorithm is $\mathcal{O}(nm)$.

4.4 Smith–Waterman Algorithm

Smith-Waterman algorithm [DEKM98, p. 22] solves the local alignment problem. It is very similar to Needleman–Wunsch algorithm. In this case we want to compute matrix A where $A[i, j]$ is the best score for the local alignment of sequences $x_1 \dots x_i$ and $y_1 \dots y_j$. The only difference between the recurrent formula in Smith–Waterman algorithm and the previous algorithm is that we are allowed to clip some prefix and suffix of the alignment. Basically, we can choose when we start the alignment. The trivial cases are:

$$A[i, 0] = 0, 0 \leq i \leq n \tag{4.3}$$

$$A[0, j] = 0, 0 \leq j \leq m \tag{4.4}$$

The recurrence relation for $1 \leq i \leq n \wedge 1 \leq j \leq m$ is:

$$A[i, j] = \max \begin{cases} 0 \\ A[i-1, j-1] + s(x_i, y_j) \\ A[i-1, j] - 1 \\ A[i, j-1] - 1 \end{cases}$$

The final result of the algorithm is the maximum value in the matrix.

The time and space complexity of this algorithm is again $\mathcal{O}(nm)$.

4.5 Seed and Extend

The time complexity of both of the algorithms is $\mathcal{O}(nm)$. This is quite slow because we usually want to work with sequences where n and m are in the order of 10^6. (The size of human genome is approximately $3 \cdot 10^9$ bp.) In the case of such sequences we cannot use an exact algorithm. We will have to make a trade-off between the precision of the algorithm and the time complexity.

Various heuristics are used to tackle this problem. The most used paradigm is seed and extend [Har07, chap. 2]. We can split this strategy into the following phases:

1. **Finding seeds (matching k-mers)** - Find all matching k-mers between input sequences x and y. These k-mers are also called seeds.
2. **Extension without gaps** - Extend all seeds in both directions to get longer alignments without using gaps. Gaps are the dashes in the alignment. We also want to connect two seeds if they are close to each other to obtain a longer alignment without using gaps.
3. **Extension with gaps** - Extend all alignments using gaps to chain alignments that are close to each other.
4. **Dynamic programming** - Use exact dynamic programming to extend the alignment (one of the two mentioned algorithms above) and improve the score.
5. **Output** - Output the highest scoring alignments.

The important part in the first phase is to choose the right length of the seed. When we take seeds that are too long, we might miss many important alignments. On the other hand, when we choose seeds that are too short we will end up with too many matching k-mers (hits) which causes us to perform too many extensions in the next phase, degrading time performance.

In the second and third phase, we keep a significance score for every local alignment. The alignments are being extended until we notice that the score is rapidly decreasing and we will not improve much by extending it more. We also have some threshold for the minimum score that we want to reach in every phase to filter out the alignments that are not significant. We probably would not manage to extend them in the next phase. This also decreases the time that is needed to spend in the next phase.

4.5.1 Our approach to finding seeds

The classical approach would be to take the most probable sequence from our model and use all the kmers of this sequence for seeding. The problem is that then most probable sequence contains many errors. Our approach is to use many sampled sequences from the conditional probability and find similar regions inn these sequences and use these regions for seeding. Using more samples might help us to eliminate the errors which occur in the Viterbi sequence.

Chapter 5

Implementation And Experiments

We can split our code into three parts: preprocessing and the data set analysis scripts, the code for hidden Markov model, and the analysis pipeline for the sampled sequences from our HMM. In this chapter, we explain the design of all three parts and then the main problems that we had to deal with. In the last section, we give an overview of the libraries that we have used in our work.

5.1 Preprocessing and the Data Set Analysis

In this section, we introduce our data set and show some basic statistics about it. Afterwards, we explain what subset of the whole data set we used for our experiments.

5.1.1 BWA-MEM

In order to find out what is the quality of our reads, we needed to align them to the reference sequence. **The reference sequence** is believed to be the true DNA sequence of the organism that was used in the sequencing. At first, we align each read to the reference sequence and then we keep the part of the reference sequence corresponding to each read. We will call the extracted part of the reference sequence a **reference read**. We assume that this is the read that we should have obtained if the sequencing was error-free.

We used `BWA-MEM` [Li13] for this task. `BWA-MEM` is a widely used tool for mapping sequences against a large reference sequences. Running `BWA-MEM`, we have used the flag `-x ont2d` which turns on optimizations for Oxford nanopore reads. `BWA-MEM` is routinely being used in many nanopore software tools, including Nanopolish [LQS15], Deeepnano [BBV16], Nanocall [DDY$^+$16] and MarginAlign [JFM$^+$15].

When the above mentioned flag is not used with our data, we can notice that the aligner often does not manage to find any alignment of the query sequence against the reference sequence. The problem with mapping nanopore reads is that these reads have very high error rate. `BWA-MEM` algorithm is based on the seed and extend approach which we described in the section 4.5. It uses maximum exact matches (MEM) for seeding and Smith-Waterman algorithm (section 4.4) for extending the seeds. The important part for us is that it performs local alignment so that it clips some parts of the sequence from both of the ends.

`BWA-MEM` takes as the input FASTA file which is a very simple text file format. For example, it looks like this:

```
>name of the sequence
ACTGAAAATAGATAAAGTAG
```

The output format of this tools is more complex. It can output `SAM` or `BAM` file. `SAM` file is a text file and `BAM` is a binary file which can be in a compressed or non-compressed format. We worked with `SAM` output. Unfortunately, SAM format is not very readable and also not so straightforward to parse. The alignment is not stored in the format as shown in the Chapter 4, but written as a compressed string. We used pysam (`https://pysam.readthedocs.io/en/latest/`) library for parsing the SAM files. This library is a python wrapper over htslib (`http://www.htslib.org/`) written in `C`.

SAM format defines more operations which can be used in the alignment. Besides match, mismatch, deletion and insertion it additionally allows skipping, padding, and clipping. In our experiments, we have only observed clipping. The meaning of these operations is not strictly defined in the documentation for BWA-MEM. SAM format also differentiates between two types of clipping – soft clipping and hard clipping. The difference is that the hard clipped part is not included in the resulting SAM file. We also noticed that hard clipping is used only for couple reads.

From this we defined two metrics for measuring identity:

$$\frac{M}{I + D + M + S + C} \tag{5.1}$$

$$\frac{M}{I + D + M + S} \tag{5.2}$$

where M is the number of matches, I is the number of insertions, D is the number of deletions, S is the number of mismatches (substitutions) and C is the length of the clipped part of the sequence.

The output sometimes contain multiple different alignments. We always chose the alignment with the greatest number of matches.

Mapping one read to the reference genome takes around one second. We worked only with template strand (data set of 27076 reads). Since our data set is so large we decided to parallelize many scripts that we used for analysis. We used `make` to run things in parallel with flag `-j number_of_jobs`, which tells `make` to split the data into multiple processes and do it in parallel.

5.1.2 FAST5

FAST5 is the file format in which the reads from MinION are stored. This file format is build on the top of HDF5 - hierarchical data format designed for storing large data sets. Oxford nanopore technologies customized this data format for storing the nanopore reads.

There are many libraries for working with this data format in many languages. We used `python` , `C++` , and `BASH` for accessing these files. Very handy GUI for viewing these files is `hdfview`. For `C++` , we have used a library specifically for FAST5 file format written by Matei David which can be found on github: `https://github.com/mateidavid/fast5.git`. For `python` , we used `h5py` library. In `BASH` scripts we used poretools [LQ14] – command line tool which is written in `python2` and `R` and allows basic analysis of nanopore reads, including plotting basic graphs summarizing the data.

For our purpose, the interesting parameters from FAST5 file are:

1. Parameters of k-mer models (Gaussian parameters) - emission probabilities in HMM.
2. Scaling parameters for Gaussian distributions.
3. Event sequence for sampling algorithm, Viterbi algorithm, and training HMM.
4. Base called sequence. This file contains three sequences – complement, template, and 2D basecall. We needed these sequences for aligning them to the reference sequence.

5.1.3 Data Set

For our experiments, we used a data set from *Escherichia coli K-12 MG1655* which can be downloaded from these two links:
`https://www.ebi.ac.uk/ena/data/view/ERX985671`,
`http://trace.ddbj.nig.ac.jp/DRASearch/experiment?acc=ERX985671`.

For simplicity, we used only template strands for our experiments. First, we tried to map these reads to the reference sequence by using BWA-MEM as we mentioned in Section 5.1.1. The reads which did not map to the reference sequence were immediately discarded. Then we looked at the move lengths that Metrichor predicted for the template strands. We split our data into two groups: the reads for which Metrichor predicted moves lower or equal to two (25162 reads) and reads for which it predicted at least one move longer than two (1911 reads). Afterwards, we randomly sampled from both of the data sets 1000 reads. Most of our analysis was done on the first data set. The second data set was only to find out whether our HMM performs significantly worse on these kind of data since we allow only skips of length one in our model (see Section 5.2).

Some basic statistics of the whole data set are presented in Figure 5.1 and Table 5.1. The read length influences the runtime and memory consumption of our algorithms. MinION can sometimes sequence reads of length even 40000. Also notice between the difference between the identity defined by 5.2 and by 5.1.

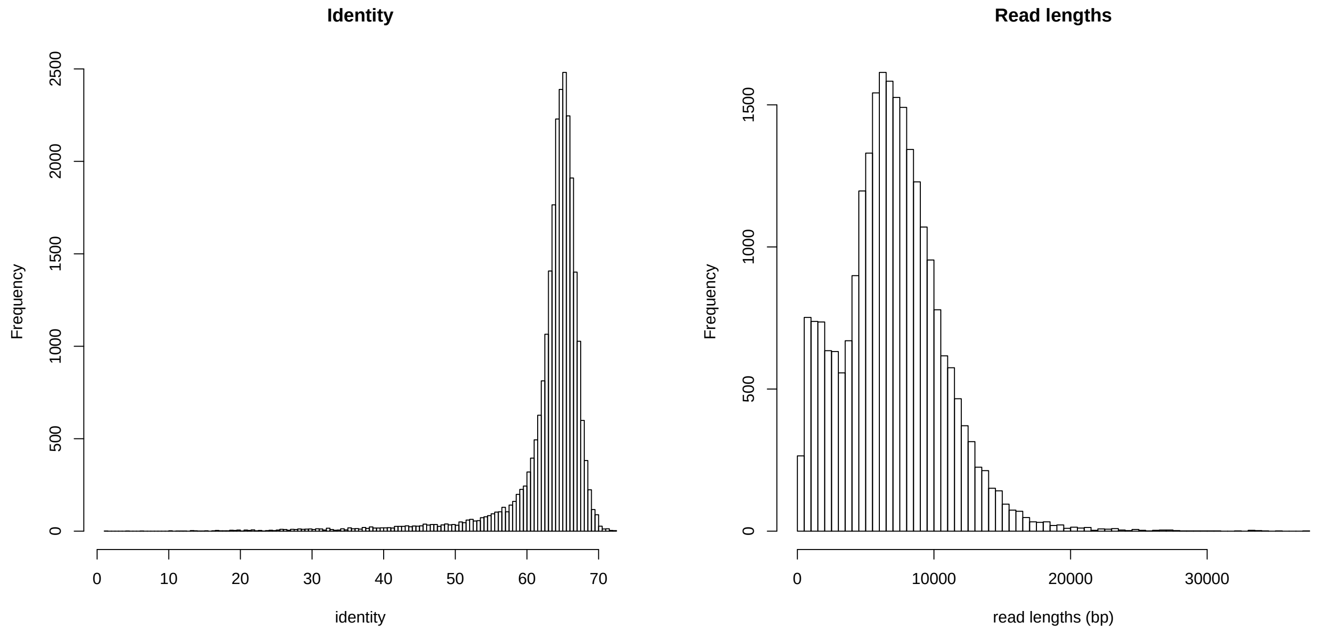

Figure 5.1: Identity (%) defined by Equation (5.2) and read lengths (bp) for the whole data set.

	Min	1st Quantile	Median	Mean	3rd Quantile	Max	Std
Identity 5.2 (%)	1.12	62.76	64.58	63.08	65.89	72.47	1.72
Identity 5.1 (%)	58.82	64.60	65.67	65.71	66.77	84.42	6.01
Read length (bp)	230.00	4689.00	6825.00	6967.00	9060.00	37270.00	3646.39

Table 5.1: Basic statistics of the whole data set.

5.2 Hidden Markov Model Implementation

We have implemented algorithms related to HMMs in `C++` . We split the whole task into two binaries: training (`train_move_hmm_main`) and, base calling and sampling (`sample_move_hmm_main`). After the training phase, the HMM is serialized into JSON file. We chose this file format because it is quite readable and nearly every programming language has a library for working with JSON format. In `C++` , we used `jsoncpp` library.

We designed a template class for HMM which implements methods for Viterbi algorithm described in Section 2.5 and the sampling algorithm described in Section 2.7. Both of the algorithms are implemented to work with silent states. The idea was to have a more general class that would serve as an abstraction above both of the algorithms that we want to use. This class takes the specific details of the architecture of HMM as arguments – the list of transitions for every state and the list of states with probability distributions. The template argument for this class is the emission type. Now, we are working only with the mean current levels for the events but in future, we might decide to use also the other parameters of the events such as the standard deviation of the current.

Transition is a `struct` storing transition probability and the node that the transition leads to. We also have an abstract class for states (`State`) and classes for individual states (`SilentState` and `GaussianState`) which inherit from the abstract class.

We used HMM architecture described in Section 3.3 and we only allowed skips of length one. The parameters for HMM were estimated by using Equation (3.4). For pseudocount we chose the value one. The problem with choosing a fixed threshold for the length of skips is that it is not clear what we should do when we encounter a transition in the training data that skips more bases than the threshold. It is very rare but it might happen so we chose a very simple strategy to handle this problem. We just do not count this kind of transitions and move on to the next transition as if nothing has happened.

We split our data set into two parts – training and testing set. We randomly shuffled our data set and chose the reads for which the number of bases sums up to approximately 70% of all bases in the data set. Training, running Viterbi algorithm, and sampling 250 samples for every read took approximately five hours. We decided no to parallelize this part, because it required a lot of memory for computing the forward probability matrices for long reads.

5.2.1 Arithmetics with Very Small Numbers

The probabilities of paths in the HMM can be very small. For the sequences of lengths of 10^4 bases the probabilities drop to very small numbers such as 2^{-5000} which brings problems with representing these numbers. The most precise data type for floating point numbers in `C++` is `double`. It uses 11 bits for the exponent. Therefore the smallest number that can be represented is $2^{-2^{10}-1} = 2^{-1023}$ and also mantissa is limited to 52 bits. The usual approach to this problem is to transform these numbers to logarithmic scale. For example: instead of storing number 0.5, we store number $\log_2(0.5) = -1$. We hold these numbers in this representation all the time so if the intermediate value was a number like 2^{-2000} then in the standard 64-bit floating number representation it is equal to zero but in our representation it is -2000 which can be stored in `double` or `float`. This approach was inspired by the article about implementing numerically stable HMM [Man06].

Now, when we are working with log-transformed values the multiplication changes to addition because:

$$\log_2(a \cdot b) = \log_2(a) + \log_2(b) \tag{5.3}$$

The instruction for adding two numbers is also faster than multiplication which is another advantage of this representation.

Division can be implemented also in the similar way by using this law of logarithm:

$$\log_2\left(\frac{a}{b}\right) = \log_2(a) - \log_2(b) \tag{5.4}$$

Again we change division into subtraction which is also more efficient than division.

Unfortunately, the addition of two log-transformed numbers is not as easy. Let us say that we have numbers $a' = \log_2(a)$ and $b' = \log_2(b)$. We want to calculate $\log_2(a+b)$ using a' and b'. Then by factoring out b and applying Equation (5.3) we obtain:

$$\log_2(a+b) = \log_2\left(b\left(1+\frac{a}{b}\right)\right) = \log_2(b) + \log_2\left(1+\frac{a}{b}\right). \tag{5.5}$$

We need to express Equation (5.5) in terms of a' and b'. We use the fact that exponential function is inverse to the logarithm. In our case, we know that: $a = 2^{a'}$ and $b = 2^{b'}$. By using these two equations, we simplify our expression into the form:

$$b' + \log_2\left(1+\frac{2^{a'}}{2^{b'}}\right) = b' + \log_2\left(1+2^{a'-b'}\right). \tag{5.6}$$

In summary, we derived the following identity:

$$\log_2(a+b) = b' + \log_2\left(1+2^{a'-b'}\right). \tag{5.7}$$

During implementing we can encounter two additional problems. The first problem is the logarithm of zero. `C++` returns `-HUGE_VAL` for $\log_2(0)$. This also needs to be handled separately in the multiplication and division.

The second problem is more hidden. We can basically compute $\log_2(a+b)$ in two ways by using the eq. (5.7) as

$$b' + \log_2\left(1+2^{a'-b'}\right) \tag{5.8}$$

or as

$$a' + \log_2\left(1+2^{b'-a'}\right) \tag{5.9}$$

From mathematical point of view those two equations are equivalent but in `C++` they are not. The order of a' and b' in the exponent does matter. For example, let us take the number 10^{-350} which is lower than 2^{-1023} and 10^{350} which is greater than 2^{1023}.

Both of the numbers cause an overflow in `C++` . The number 10^{-350} is truncated to zero and the number 10^{350} is evaluated as a special constant `inf` which stands for infinity. Therefore $\log_2(1 + 10^{-350})$ is zero and $\log_2(1 + 10^{350})$ is inf. Let us say that we have $a' = 2000$ and $b' = 100$. Then when we use the Equation (5.8) we end up with infinity and in the case of using the Equation (5.9) we get $a' = 2000$ which is not completely accurate but still better than infinity. If $a' > b'$ then we want to use the Equation (5.9) otherwise we want to use the Equation (5.8).

We implemented this part of code in `C++` which allows us to overload operators for classes. Therefore we wrote a class `Log2Num` which has overloaded operators for addition, multiplication, division, comparison and the operators for writing into the streams (<<, >>). This abstraction allows us to hide all the implementation details and problems with the logarithm of zero.

5.2.2 Encoding k-mers

Another problem that we encountered was encoding k-mers into integers. In some situations, it is more convenient to work with integers than strings. Additionally, strings take more space than integers. For example, we need to map k-mers into the state ids of the HMM for the dynamic programming.

We can see every k-mer as a number in the base-4 numeral system. More formally, we can define homomorphism:

Definition 9. *Homomorphism* $h : \{A, C, T, G\}^* \to \{0, 1, 2, 3\}^*$ *maps* $A \to 0$*,* $C \to 1$*,* $T \to 2$*,* $G \to 3$*.*

We can use the classical algorithm for conversion between numeral systems and convert this base-4 number into decimal system. Now, we can work with it as an integer in our code. When we want to convert this number into a string then we can again use the classical algorithm for conversion between bases and get a number in base-4. The only thing that we have to keep in mind is that normally when we work with numbers we do not care about leading zeros. In our case, we cannot ignore those zeros. Therefore after decoding, we have to add zeros to the beginning to get a string of length k.

5.2.3 The Translation of State Sequences into DNA Sequences

Both of our algorithms for HMM return a sequence of states. Afterwards, we need to translate it into DNA sequence. The main problem of the translation is that it is not unique because of loops and skips in the model. For example, when we have two consecutive states for 5-mers `TCGCG` and `CGCGA` then we might interpret it as a move of length one or move of length three. Depending on that we will end up with a sequence

```
$N = length(states)$
$previous\_kmer = states[1]$
$result = previous\_kmer$
for $i = 2, 3, \dots, N-1$ do
    $next\_kmer = kmerToString(states[i])$
    $move = shortestMove(prev\_kmer, next\_kmer)$
    $result$ += the suffix of length $move$ of the string $next\_kmer$
    $prev\_kmer = next\_kmer$
end for
return result
```

Figure 5.2: Translation of the state sequence to DNA sequence. The input of the algorithm is the array *states*. This array is indexed from zero but the first state is the initial state.

`TCGCGA` or `TCGCGCGA`. Our algorithm always chooses the smallest possible move, in this case `TCGCGA`. Notice that when we have transition from `AAAAA` to `AAAAA`, we always choose the move of length zero. For example, if the sequence of states looks like this:

`AAAAA, AAAAA, AAAAA, AAAAA, AAAAA, AAAAA, AAAAA, AAAAA, AAAAA, AACTG`

then it will be translated to `AAAAACTG`. Another consequence of this heuristics is that we will never end up with homopolymers longer than five.

Pseudocode of this algorithm is in Figure 5.2.

5.3 Pipeline For Analyzing Samples

The output from the HMM (`sample_move_hmm_main`) are separate files for each read that we used for testing. Each file contains Viterbi sequence, empty line and the sequences for samples. Every sample is on separate line. In the following section, we describe analysis that we have performed on this data.

5.3.1 Base Calling

First, we need to estimate the quality of the output from our HMM. We decided to compare the most probable sequence from our model with the Metrichor base called sequence. We also wanted to find out how different are our samples from these two sequences.

We aligned Viterbi sequence, Metrichor sequence, and the samples to the reference read by `BWA-MEM` with flag `-x ont2d` and compared identity percentage between reference and all of these sequences. All samples were successfully mapped to the reference sequence. See the two graphs with boxplots in Figure 5.3.

When we look at the statistics calculated by Equation (5.1) from the whole testing set, we notice that the mean identity of Metrichor is 65.7% and median is 65.88%. The mean identity of our model is 62.47% and 64.06%. On the other hand, when we look at the values without clipping then the differences are smaller. The mean identity of Metrichor is 66.45% and median 66.33%. The mean identity of our model is 64.87% and 64.84%.

When we evaluate the identity on the second data set we notice only 1.97% difference between Metrichor and Viterbi without clipping. With clipping, the difference is 5.88%. Metrichor predicted for these reads that they contain moves longer than two and our model does not allow moves longer than two. Therefore, we can say that this is a difficult data set for our model. It seems that it can still perform quite well.

Our main goal is not basecalling therefore it is not so important to have the best model for the data. It is just an intermediate step in the design of our algorithm. For basecalling, recurrent neural networks (RNN) achieve better results [BBV16] and even Oxford nanopore technologies decided to move from HMM to RNN in the most recent release.

5.3.2 Comparison of k-mer Sets

Our primary goal is to find out what are the appropriate lengths of k-mers that we can use for seeding our alignment algorithm and also what is the reasonable number of samples that we should use. These are the two main parameters for our alignment algorithm.

Definition 10. k-mer of a sequence x
We call a sequence y a k-mer of a sequence x if and only if $|y| = k$ and y is a substring of the sequence x.

We decided to compare sets of k-mers between the reference read, Viterbi sequence, Metrichor sequence, and samples. Let R be the set of all k-mers of the reference read and S_i be the set of all k-mers of the first i samples. Then we are mostly interested to find out what is the size of $R \cap S_i$. We will sometimes use the standard terminology from the statistics and denote all k-mers in $R \cap S_i$ as true positives, all k-mers in $R \setminus S_i$ as false negatives and so on. Notice Venn diagrams in Figure 5.4.

We decided to try all $k \in \{9, \ldots, 30\}$. For example, `BWA-MEM` [Li13] uses seeds of minimum length 19, `BLAST` [AMS+97] uses seeds of minimum length 11 and `bowtie` [LTP+09] uses seeds of length 28.

We chose to sample 250 sequences from our HMM. We do not want to choose too many because it would slow down the algorithm but we also do not want to choose very

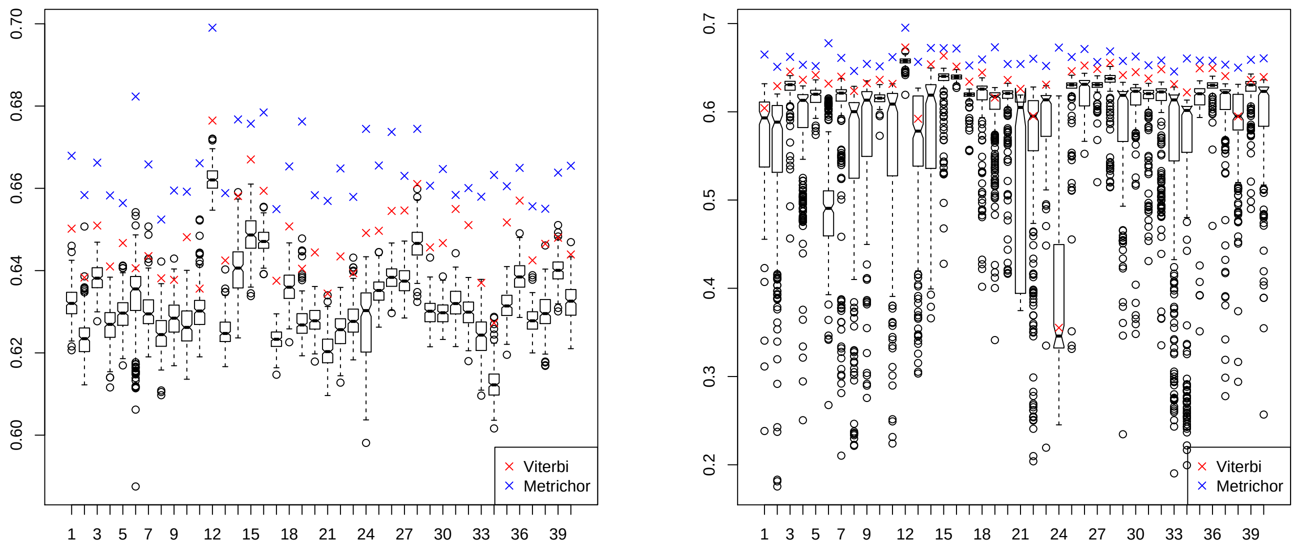

Figure 5.3: Boxplots of percent identity between the reference and samples. Each boxplot corresponds to an individual read (in total, we show 40 reads). Red and blue crosses correspond to Viterbi and Metrichor sequence as you can see in the legend. The circles are outliers. The identity in the left plot is defined by the Equation (5.1) and in the right plot by the Equation (5.2). Notice in the left plot that the difference in identity between our basecalled sequence and the Metrichor is only 2% which is a very good result. In the right plot we also took into account the clipping which changed the boxplots a lot. Metrichor and Viterbi sequence are still close to each other but the boxplot for samples is more stretched – the difference between upper and lower boundary is greater and we also have more outliers. That means the problem with the alignment of samples to the reference read is that quite a large part from the ends is clipped quite often. Viterbi sequence is usually somewhere around the upper boundary but sometimes it can be found also in the interquartile region.

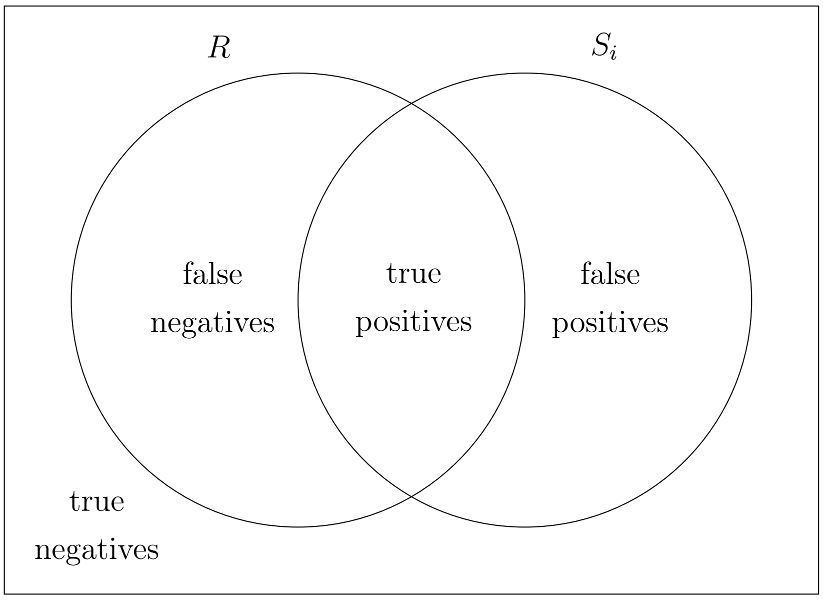

Figure 5.4: Venn diagrams for k-mer sets

few because we would not find enough k-mers matching the k-mers in the reference sequence.

First, we need to compute R and S_i and then compute the intersection. We need to do it efficiently because we need to run this algorithm for all k-mer lengths, for all samples, reference read, Viterbi sequence and also Metrichor sequence.

The main idea is based on the rolling hash algorithm which is used, for example in Rabin-Karp algorithm for string searching [Wik16d]. Let us start from the beginning of a sequence $a = a_1 a_2 \dots a_n$ and take the first k-mer $a_1 a_2 \dots a_k$. We can encode this k-mer into an integer (hash) in the same way as in Section 5.2.2 and then we want to encode the next k-mer $a_2 \dots a_{k+1}$ and so on. The idea that makes our approach efficient is that we can reuse the code of the previous k-mer in order to calculate the code of the next k-mer.

When we have a k-mer $a_0, \dots, a_{k-1}$ then it can be represented in the decimal as a number

$$x = a_0 \cdot 4^{k-1} + a_1 \cdot 4^{k-2} + \dots + a_{k-1} \cdot 4^0 \tag{5.10}$$

The code of the next k-mer is:

$$y = a_1 \cdot 4^{k-1} + a_2 \cdot 4^{k-2} + \dots + a_{k-1} \cdot 4^1 + a_k \cdot 4^0 \tag{5.11}$$

Therefore we can calculate y more efficiently as:

$$(x - a_0 \cdot 4^{k-1}) \cdot 4 + a_k \cdot 4^0 \tag{5.12}$$

The important thing to notice is that we need $2 \cdot k$ bits to encode a k-mer. Therefore 32-bit integer is not sufficient for us.

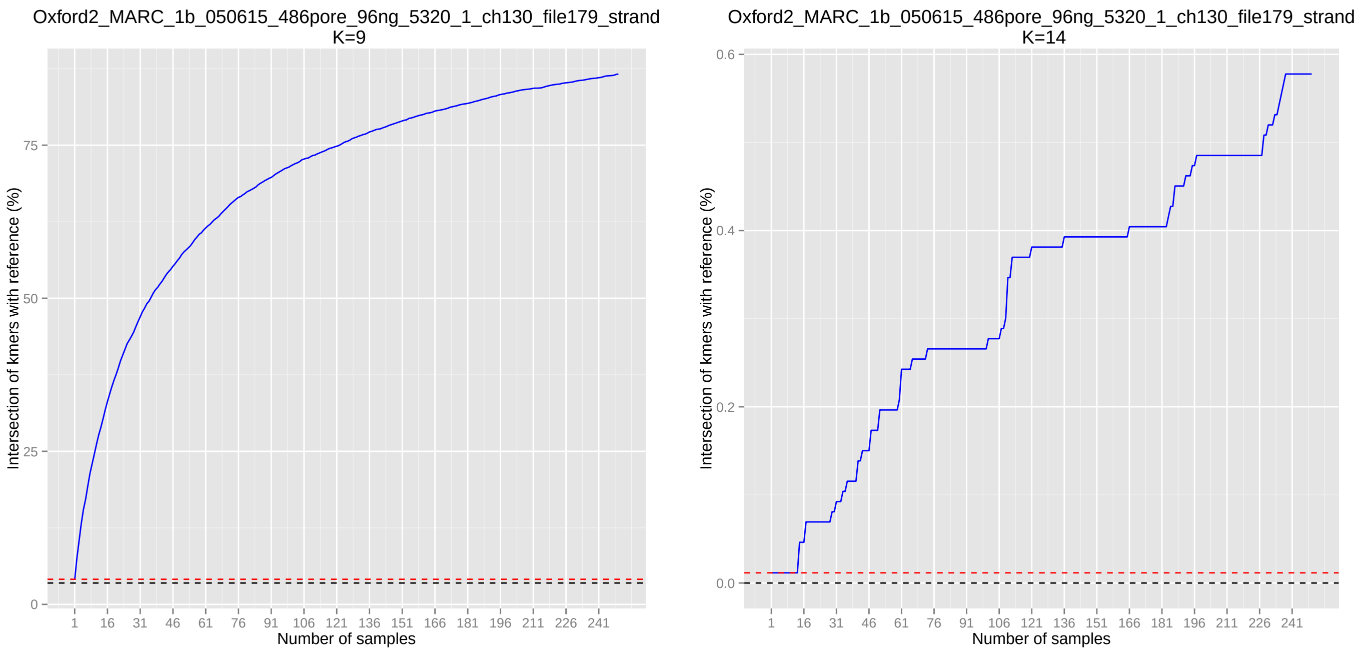

Figure 5.5: A sample plots for a random read and $k = 9$ and $k = 14$. The highest sensitivity for $k = 9$ grew to 85%, while for $k = 14$ it only reached 0.6%. The red dashed line is the sensitivity of the Viterbi sequence from our model and the black dashed line is the sensitivity of Metrichor sequence.

This algorithm can be implemented using iterator design pattern. We have a class that that represents the current k-mer window in the sequence. This class provides methods `next()`, `currentkmerCode()` and `currentkmer()`. Method `next()` just slides the window by one to the right or returns -1 if we are at the end of the string.

From now on, we just work with the codes of k-mers. For storing these codes, we use `std::set<long long>()`. In the first step, we compute set R and then we iterate over all samples. To iterate over a particular sequence of a sample we use our k-mer iterator. We maintain set S_i. If a new element was inserted into S_i then we check if this element is in R. If yes then we increase the counter for true positives $(R \cap S_i)$. After computing R, S_i and the number of true positives, we can compute the sizes of all four partitions which you can see in **??** in a straightforward way.

From this we can define sensitivity (hit rate) as:

$$\frac{|R \cap S_i|}{|R|} \tag{5.13}$$

The plots of sensitivity for every read are included in the appendix. Now, we only include a random sample reads to explain these plots. We have two types of plots – separate plots for every read and every length of k-mer (Figure 5.5) and plots separate for every read with all curves for lengths of k-mers in it (Figure 5.6).

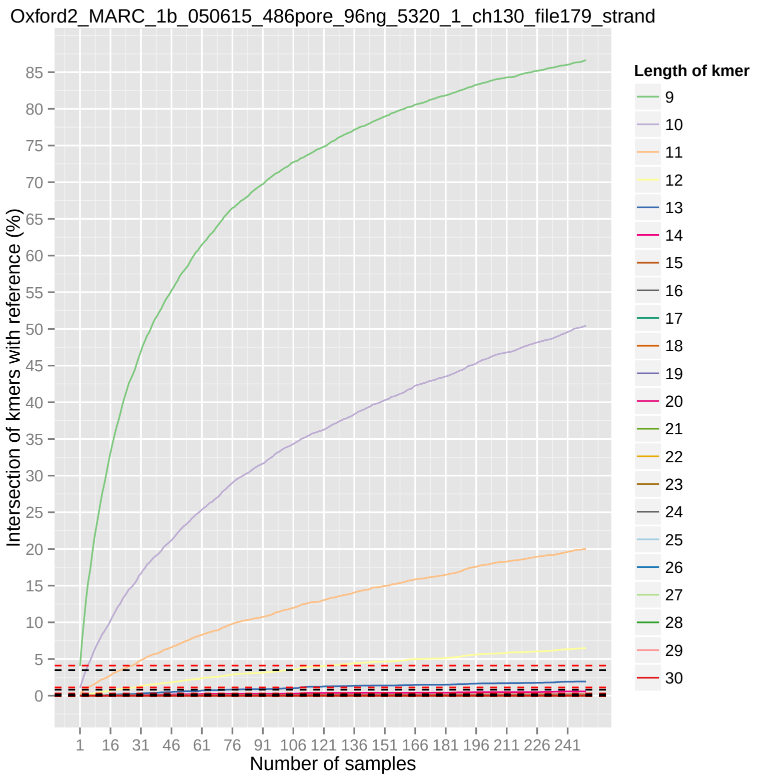

Figure 5.6: A sample plots for a random read and all $k \in \{9, \ldots, 30\}$. Every curve corresponds to different k. The red dashed lines correspond to Viterbi sequence and the black dashed lines correspond to Metrichor sequence. Both Viterbi and Metrichor match less than 5% of k-mers of the reference read. For many reads, we can notice that the curves for $k > 12$ are below 15%. In this case, it is even below 5%.

From these plots we could notice that Viterbi and Metrichor are usually around 5% or even lower. The curves for samples very quickly intersect the lines for Viterbi and Metrichor which means that we probably do not need so many samples for our algorithm.

Based on these graphs we came up with compound statistics for all reads which are very similar to standard statistical quantities. Let C_i be then number of k-mers in the read i then:

$$\text{sensitivity} = \frac{\sum_i |R \cap C_i|}{|R|} \tag{5.14}$$

$$\text{precision} = \frac{\sum_i |R \cap C_i|}{|R|} \tag{5.15}$$

$$\text{specificity} = \frac{\sum_i |(R \cup C_i)^c|}{\sum_i |(R \cup C_i)^c \cup (C_i \setminus R)|} \tag{5.16}$$

$$\frac{\text{found } k\text{-mers}}{\text{reference } k\text{-mers}} = \frac{\sum_i |C_i|}{|R|} \tag{5.17}$$

We computed all these quantities for all $k \in \{9, \ldots, 30\}$ and plotted graphs for them. The graphs are in the appendix.

In the compound graphs, we can notice that the sensitivity for $k > 20$ drops below 1%, therefore we do not want to such length of k. We also probably do not want to try $k = 9$ because we notice too many hits which slows down the alignment algorithm.

5.4 Libraries

Besides the libraries for working with various data formats that we mentioned above we also used a couple of other libraries.

For logging, we used `google-glog` library which open source and released on `https://github.com/google/glog.git`. It is also available in Ubuntu in the package `libgoogle-glog-dev`. This library allowed us to monitor our algorithm during the runtime. It was also very helpful for debugging.

We also used `google-gflags` for parsing the command line arguments in `C++` . It can be also found on `https://gflags.github.io/gflags/`.

We decided to properly test our code written in `C++` so we wrote at least some basic unit tests for every class and every function exposed in the header files. We used `https://github.com/google/googletest.git` library for this purpose. It is a very convenient unit testing framework. We used package `libgtest-dev` in Ubuntu with version `1.7.0`. It is important to used this version or a later version because some features might not work that we used and the unit tests might not even compile.

Conclusion

The goal of this thesis was to design a new approach for alignment of nanopore reads. A common approach is to translate the sequences of electric current levels into DNA sequences and then align them using a general-purpose algorithms. The algorithms used for translation of sequence of electric current levels (base calling) are quite inaccurate which is also demonstrated in our experiments. Due to this inaccuracy, a lot of information is lost in the base calling process and the alignment algorithm has to work only with this reduced information. Instead, we proposed to merge these two steps and use the raw underlying data from electric current in our alignment algorithm.

The usual approach for base calling nanopore reads is to design a hidden Markov model and then take the most probable sequence from the HMM as the base call. As we observed, this sequence contains many errors. Therefore, instead of a single optimal sequence, we have decided to use a set of suboptimal sequences sampled from the conditional distribution defined by the base calling HMM.

Many commonly used alignment algorithms use seed and extend paradigm (Section 4.5). These algorithms use only the most probable sequence from HMM for finding seeds. Our approach is to use a set of randomly sampled sequences, find similar regions among these sequences, and use these regions for seeding our algorithm.

In this thesis, we have designed two different architectures for hidden Markov model (Chapter 3). In our experiments, we found out that the first HMM is comparable the Metrichor basecall. The difference is around 2 percentage points which is quite sufficient for our purposes since our final goal is not base calling. The Metrichor model is probably more complicated because it can predict moves longer than two but still the difference in precision of our model and Metrichor is not so large.

After running our HMM, we have analyzed the samples from the conditional distribution. We compared the sets of k-mers in the samples and the reference read. We experimented with various lengths of k-mers and numbers of samples. From these experiments, we were able to identify some potentially good lengths of k-mers. It also seems that we do not need to sample that many sequences for the alignment algorithm.

In our experiments, we did not take into account positions of kmers. Therefore in future, we might want to investigate the positions of kmers and find out if there are any interesting common subsequences between the samples. In this way, we might be able to find some good chains of kmers which occur in the samples in increasing order.

Bibliography

[AMS$^+$97] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[BB11] Tomáš Vinař Broňa Brejová. *Methods in bioinformatics*, volume 1. Knižničné a edičné centrum, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Mlynská dolina, 842 48 Bratislava, 1 edition, 2011.

[BBB$^+$] JM Bernardo, MJ Bayarri, JO Berger, AP Dawid, D Heckerman, AFM Smith, and M West. Generative or discriminative? getting the best of both worlds.

[BBV16] Vladimír Boža, Broňa Brejová, and Tomáš Vinař. Deepnano: Deep recurrent neural networks for base calling in minion nanopore reads. *arXiv preprint arXiv:1603.09195*, 2016.

[CP03] Simon L Cawley and Lior Pachter. Hmm sampling and applications to gene finding and alternative splicing. *Bioinformatics*, 19(suppl 2):ii36–ii41, 2003.

[DDY$^+$16] Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C Boutros, and Jared T Simpson. Nanocall: An open source basecaller for oxford nanopore sequencing data. *bioRxiv*, 2016.

[DEKM98] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[GVM03] Valer Gotea, Vamsi Veeramachaneni, and Wojciech Makałowski. Mastering seeds for genomic size nucleotide blast searches. *Nucleic acids research*, 31(23):6935–6941, 2003.

[Har07] Robert S Harris. *Improved pairwise alignment of genomic DNA*. ProQuest, 2007.

[JFM$^+$15] Miten Jain, Ian T Fiddes, Karen H Miga, Hugh E Olsen, Benedict Paten, and Mark Akeson. Improved data analysis for the minion nanopore sequencer. *Nature methods*, 12(4):351–356, 2015.

[Li13] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[LQ14] Nicholas J Loman and Aaron R Quinlan. Poretools: a toolkit for analyzing nanopore sequence data. *Bioinformatics*, 30(23):3399–3401, 2014.

[LQS15] Nicholas James Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *bioRxiv*, page 015552, 2015.

[LTP$^+$09] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biol*, 10(3):R25, 2009.

[Man06] Tobias P Mann. Numerically stable hidden markov model implementation. *An HMM scaling tutorial*, pages 1–8, 2006.

[Mou04] David W. Mount. *Bioinformatics: sequence and genome analysis.* Cold Spring Harbor Laboratory Press, 2004.

[Nán10] Michal Nánási. Biological sequence annotation with hidden markov models. *Master1s thesis*, 2010.

[Sch16] Amanda Schaffer. Nanopore sequencing, 2016. [Online; accessed 30-April-2016] Available from `http://www2.technologyreview.com/news/427677/nanopore-sequencing/`.

[Sim16] Jared T. Simpson. Simpson lab blog, 2016. [Online; accessed 27-January-2016] Available from `https://simpsonlab.github.io/`.

[Tec] Oxford Nanopore Technologies. Oxford nanopore sequencing technologies specifications. [Online; accessed 26-January-2016] Available from `https://nanoporetech.com/community/specifications`.

[Wik16a] Wikipedia. De bruijn graph, 2016. [Online; accessed 26-January-2016] Available from `https://en.wikipedia.org/wiki/De_Bruijn_graph`.

[Wik16b] Wikipedia. Dna, 2016. [Online; accessed 30-April-2016].

[Wik16c] Wikipedia. Generative model, 2016. [Online; accessed 30-April-2016].

[Wik16d] Wikipedia. Rabin–karp algorithm, 2016. [Online; accessed 15-May-2016] Available from `https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm`.

[ZB08] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Appendix A - Implementation and Graphs

This thesis includes an attached CD, containing the source code and graphs mentioned in the text. The source code is also available at `https://github.com/rasto2211/nanopore-read-align.git`.