Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Algorithms for high-throughput sequencing data

(PhD thesis)

Vladimír Boža

Thesis advisor: Mgr. Tomáš Vinař, PhD. Bratislava, 2017

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Algorithms for high-throughput sequencing data

PhD thesis

Study Program: Informatics
Branch of Study: 9.2.1 Informatics
Supervisor: Mgr. Tomáš Vinař, PhD.
Bratislava, 2017

Vladimír Boža

88351799

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Mgr. Vladimír Boža
Študijný program:	informatika (Jednoodborové štúdium, doktorandské III. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	dizertačná
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Algorithms for high-throughput sequencing data
Algoritmy pre spracovanie dát vysokoparalelného sekvenovania

Cieľ: V súčasnosti existuje niekoľko nových technológií sekvenovania DNA a ďalšie sú vo vývoji. Kým niektoré technológie produkujú krátke a presné čítania, výsledkom iných sú veľmi dlhé čítania s chybovosťou až do 30%. Cieľom tejto práce je vytvoriť nové metódy pre kombináciu čítaní z rôznych technológií za účelom zoskladania genómu a pre spracovanie dát z technológie MinION, čo je jedna z najnovších sekvenovacích technológií. Cieľom je vytvoriť efektívne a presné softvérové nástroje použitím metód tvorby efektívnych algoritmov, pravdepodobnostného modelovania a strojového učenia.

Školiteľ:	doc. Mgr. Tomáš Vinař, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	21.02.2013

Dátum schválenia: 21.02.2013

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....

študent školiteľ

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

. . . .

Acknowledgments

I am deeply grateful to my supervisor Mgr. Tomáš Vinař PhD. for his invaluable help and guidance. I would also like to thank my wife Maria and my son Michal for their great support.

Abstrakt

V tejto dizertačnej práci sa zaoberáme problémami súvisiacmi so zostavovaním DNA sekvencií. Našim hlavným cieľom je pracovať s rozličnými kombináciami typov čítaní, vrátane krátkych čítaní s krátkymi a dlhými medzerami, či dlhých čítaní. Navrhli sme algoritmus GAML (z angl. Genome Assembly by Maximum Likelihood – zostavovanie genómu s najväčšou vierohodnosťou), ktorý systematicky pracuje s rozličnými typmi čítaní, pričom ich vlastnosti reprezentuje pomocou pravdepodobnostných modelov. GAML optimalizuje vierohodnosť zostavenia genómu, ktorá silno koreluje s kvalitou zostavenia.

Počas vývoja GAMLu sme na narazili na niekoľko zaujímavých problémov súvisiacich s indexovaním čítaní. Vyvinuli sme novú dátovú štruktúru CR-index, čo je index pre množinu čítaní, využívajúci fakt, že čítania často pochádzajú zo spoločného nadslova. Tiež navrhujeme MH-index, pre dlhé DNA sekvencie, ktorý využíva minimalizéry.

Oxford Nanopore MinION je technológia, ktorá produkuje dlhé čítania, ktoré sú dôležité pre zlepšenia zostavenia genómu. S použitím rekurentných neurónových sietí, sme vyvinuli nástroje, ktoré zlepšujú preklad elektrického signálu zo sekvenátora na DNA bázy a taktiež sme zlepšili algoritmy na porovnávanie signálov zo sekvenátora s referenčnou sekvenciou.

Kľúčové slová: zostavovanie genómu, indexovanie reťazcov, minimalizéry, rekurentné neurónové siete

Abstract

In this thesis, we study several problems related to the DNA sequence assembly. Our main focus is on handling different combinations of read types, including short reads with short and long inserts, and long reads. We propose the genome assembly by maximum likelood (GAML) framework, which handles a variety of sequencing data in a systematic way by using probabilistic models. In particular, GAML optimizes assembly likelihood score, which has previously been shown to be strongly correlated with the assembly quality.

During the development of GAML, we have encountered several interesting problems concerning indexing of sequencing reads. We have developed a new data structure CR-index, an index for a collection of short reads that exploits the property that reads usually originate from a common superstring. We also propose a index for long DNA strings, based on the idea of minimizers, called MH-index.

Oxford Nanopore MinION is a technology producing long reads, which are important for improving the sequence assembly. By using recurrent neural networks, we have developed tools for improving the base calling (translation of the raw electric signal from the sequencer to the DNA bases). We have also improved approaches for comparing raw signals from the sequencer to the reference sequence.

Keywords: sequence assembly, string indexing, minimizers, recurrent neural networks

Contents

Chapter 1

Introduction

Knowledge of DNA sequences has become indispensable for basic biological research and in numerous applied fields like diagnostics, forensic biology, etc. It is also important for understanding cancer, fighting antibiotic resistant bacteria, etc. With decreasing cost of the DNA sequencing we are able to sequence more organisms, but consequently we have to handle much bigger amounts of data.

For our purposes, we can consider DNA sequence to be a string over a four letter alphabet $\{A, C, G, T\}$ (individual letters will be called DNA bases). Usual length of DNA sequence of an organism is between millions and billions of characters. For example, human DNA has approximatelly three billions bases. Unfortunately, we are still not able to read the whole DNA sequence at once, we can only read it in small pieces called *reads*. Usual length of reads vary between hundred to tens of thousands of bases depending on the sequencing technology. Thus, our only option is to try to computationaly reconstruct original sequence, using overlap information between reads. This process is called *sequence assembly*.

Theoretically formulating DNA genome assembly leads usually to NP-hard problems (Kececioglu and Myers, 1995) (see more in Chapter 2), thus we resort to heuristic approaches. To make matters worse, there are also multiple technologies (Quail et al., 2012; Liu et al., 2012) for reading DNA with very different properties like read length, error rate, error types, etc. Some technologies produce paired reads with known distance between pairs. Usually assembly algorithms are tailored to a specific type of data and thus do not work with every combination of data we can get. For example, ALLPATHS-LG (Gnerre et al., 2011) assembler requires paired reads with specific distance between them. In Chapter 2, we present GAML (Genome assembly by maximum likelihood), our own genome assembly framework, which allows to seamlessly combine multiple data types. GAML is based on optimizing genome likelihood score (Ghodsi et al., 2013; Clark et al., 2013; Rahman and Pachter, 2013), which was shown to strongly correlate with the quality of the assembly. In GAML, we optimize the likelihood score by using simulated annealing, while we speed up likelihood evaluation by reusing results from the previous evaluations. We experimentally evaluate our algorithm and show that it produces comparable results to other assemblers, which are tailored to specific datasets. We also show that GAML is able to correct and improve previously created assembly by an other assembler.

Eventhough assembly algorithms are heuristics, they need to use many theoretically interesting data structures and algorithms to scale to large genomic datasets. Examples include probabilistic data structures for de Bruijn graphs (Chikhi et al., 2012), minhashing approaches for overlaps (Koren et al., 2017) and fast and memory efficient algorithms for string indexing (Li and Durbin, 2009; Ferragina and Manzini, 2000). In GAML, we encountered a problem of indexing large collections of short reads. In Chapter 3, we present our solution to this problem, a compressed index for genomic data. called CR-Index. In CR-Index, we exploit the fact that reads are usually derived from some unknown superstring, which we approximately reconstruct and use for indexing. We also achieve a large memory savings by carefully handling sequencing errors. CR-Index uses much less space that other solutions, while maintaining fast querying speed.

Another traditional problem is indexing of a long DNA string and finding positions of short substrings of prespecified size. Typical solutions include hash tables, which are fast, but use more memory, and FM-Index (Li and Durbin, 2009; Ferragina and Manzini, 2000), which uses less memory, but is slower. Inspired by success of minhashing for other genomic tasks (Koren et al., 2017; Wood and Salzberg, 2014), in Section 3.2 we present our own idea for fast and practical data structure called MH-index for searching for fixed-length small substrings in a long string, which uses minhashing. Our new data structure has fast speed comparable to hash-tables and uses small memory amount of memory comparable to FM-indices.

In Chapter 4 we explore problems relate to Oxford Nanopore MinION, which is one of the newest sequencing technologies showing a great promise in clinical applications and other areas. MinION produces long but very noisy reads. An interestring part of MinION from computer science point of view is that it does not perform exact base detection directly in hardware, but instead only collects electrical signal, which needs to be translated into DNA bases using a special software called base caller. We present our own base caller for MinION, called DeepNano, which uses recurrent neural networks. On older versions of MinION, DeepNano had much better accuracy and base calling speed than the original base caller. On newer versions, DeepNano has a slightly worse accuracy, but slightly better speed than the original base caller.

One interesting ability of MinION is to produce results while DNA is being sequenced and to reject reads while they are being sequenced, so we can focus sequencing on interesting parts of the DNA (Loose et al., 2016). Unfortunately, base calling algorithms are slower than the sequencing hardware, so we either have to use large computers, or come up with fast approach for rejecting reads. One of such approaches, is working directly with electrical signal coming from MinION, using algorithms like dynamic time warping (Sankoff and Kruskal, 1983). We experimentally show that the original version of DTW is not suitable in our case and propose a suitable improvements and experimentally demonstrate that our improvements lead to better sensitivity and specificity.

In each chapter we provide provide overview of related research alongside the presented work. We present our original work in Sections 2.5 - 2.8 (GAML), Section 3.2 (MH-index), Section 3.4 (CR-index), Section 4.2 (Deepnano), and Section 4.3 (our improvement for dynamic time warping).

Chapter 2

Probabilistic Sequence Assembly

Current technologies cannot read the whole sequence at once, instead they produce many (usually uniformly sampled) overlapping substrings of the sequence, called *reads*. The length of reads ranges from hundreds of bases to tens of thousands of bases depending on the sequencing technology. The goal of the sequence assembly is to reconstruct the original string.

In this chapter, we present the genome assebly by maximum likelihood (GAML), our own framework for the genome assembly problem. Its main benefit is an ability to seamlessly integrate any combination of genome sequencing data. In contrast, most of the current genome assembly tools are tailored to specific data types.

We first introduce a theoretical background of the sequence assembly, review current algorithms and heuristics used in practice, as well as problems of evaluating the quality of the sequence assembly, and calculating the likehood of the genome assembly. Then we describe our own GAML framework, which introduces new extensions to the assembly likelihood evaluation and provides an algorithm for finding assemblies with high likelihoods.

Sections 2.1 - 2.4 give an overview of methods used in modern assemblers and give an introduction to the problem of comparing genome assemblies. Sections 2.5 - 2.8 describe and evaluate our new framework, which we published in Boža et al. (2014).

Perhaps the oldest formulation of the assembly task as a computer science problem is the shortest common superstring problem.

Definition 1 (The shortest common supersting) *Given a set of strings $\mathcal{P} = \{S_1, S_2, \ldots, S_k\}$, the shortest common superstring is the shortest string S that contains every string from $\mathcal{P}$ as a substring.*

Unfortunately, finding the shortest common superstring is NP-hard (Gallant et al., 1980; Garey and Johnson, 1979). There is a known 2.5-approximation algorithm (Sweedyk, 2000). The most common greedy heuristic for this problem can be described as follows.

```
while $\mathcal{P} > 1$ do
    $a, b \leftarrow$ two strings from $\mathcal{P}$ with the largest overlap
    $c \leftarrow merge(a, b)$
    $\mathcal{P} \leftarrow \mathcal{P} \setminus \{a, b\} \cup c$
end while
```

The approximation factor for this heuristic is not known. It is conjectured to be a 2-approximation algorithm (Blum et al., 1994), which is also the best known lower-bound for the approximation ratio. The best known upper bound is 3.5 (Kaplan and Shafrir, 2005).

In the following sections, we will examine more realistic formulation of the assembly problem.

2.1 Challenges of Real Sequencing Datasets

Over the years, it has been shown that the shortest common superstring formulation does not represent the assembly problem very well. On one hand, the problem is difficult to solve computationally, and the approximation algorithms do not yield very practical results. On the other hand, this formulation does not consider some specifics of the underlying data, which are outlined below.

Repeats. Original DNA sequence usually contains long repetitive regions, which are often longer than the longest available read. Thus, if we have multiple repeated regions in a row, the shortest common superstring solution is likely to collapse these repeats. Moreover, if there are multiple repeated regions with unique sequences between them, the shortest common superstring formulation will be unable to resolve the order of these unique sequences.

Reverse complement. The DNA comes in two strands, which are reverse complements of each other.

Definition 2 (Reverse complement) *For given DNA sequence S, its reverse complement is defined as $RC(S) = h(S^R)$, where $h(\cdot)$ is a homomorphism where $h(A) = T$,*

Technology	Read length	Error rate	Paired reads	Cost per million bases	Reads per run	Time per run
Sanger	900	0.1%	Yes	\$2400	$\approx$ 100	few hours
454	700	0.1%	Yes	\$10	1 million	one day
Illumina	50 - 300	2%	Yes	\$0.05 – \$0.15	3 billion	few days
PacBio	20000	14%	No	\$0.13 – \$0.60	50000	few hours
Oxford Nanopore	30000	10% – 20%	No	\$0.5	20000	one day

Table 2.1: **Overview of current sequencing technologies**

$h(C) = G$, $h(G) = C$, $h(T) = A$.

During sequencing, we usually obtain reads from both strands of the DNA and we do not have any information about the read orientation (which strand the read comes from).

Errors in reads. Sequencing technologies are prone to errors. Sometimes there are small errors in reads, substitutions, small insertions and deletions. The error rate depends on a specific sequencing technology. For example, some technologies have higher amount of substitutions, while others have high amount of insertions. There are also reads which do not belong to the sequenced genome, but result or contamination.

Paired reads. Some technologies produce reads in pairs for which we know an approximate distance in the genome. The total length of reads plus approximate distance between reads is called *insert size*. Paired reads with long insert sizes can help resolve ambiguities around repetitive regions.

Variety of sequencing technologies. There are several sequencing technologies currently available. They vary in cost, read length, accuracy, throughput, etc. 2.1 sumarizes currently used sequencing technologies, based on data from Quail et al. (2012); Liu et al. (2012); Mikheyev and Tin (2014). We have also added statistics of old Sanger sequencing for comparison.

A wide variety of technologies presents an additional challenge to the assembly software. Ideally, we would handle multiple combinations of read libraries from possibly different technologies (like Illumina + PacBio, or multiple libraries of Illumina reads with different insert sizes). We will discuss this topic further in Section 2.2.

Due to these problems, it is obvious that finding the shortest common superstring is not a good formulation for the sequence assembly. In practice, our goals are usually more humble and we are satisfied with reconstructing unambiguous parts of the DNA sequence. Assembly tools usually produce substrings from the original sequence called *contigs*. Sometimes it is possible to detect approximate distances between contigs, but the DNA sequence between them remains unknown. A *scaffold* is a sequence of contigs with known approximate distance between them. In genome assemblies unknown areas are often represented as a strings of several Ns.

2.2 Overview of Current Solutions

In this section, we discuss several attempts at theoretical formulation of the sequence assembly problem and then we describe currently used assembly algorithms.

Informally, our goal is to reconstruct DNA sequence from reads, which:

- are much smaller than the original sequence,
- have an unknown orientation,
- contain errors,
- some of them did not originate from the original sequence,
- and may come in pairs with approximatelly known distance.

To account for the possibility of sequencing errors, Kececioglu and Myers (1995) proposed a variant of the shortest common superstring problem.

Definition 3 (The shortest common superstring with errors) *Given set of reads $\mathcal{F}$ and error rate ε, find the shortest sequence S such that for every $A \in \mathcal{F}$ there is a substring B of S where:*

$$min(d(A, B), d(\bar{A}, B)) \leq \varepsilon|A|$$

This formulation can be also extended to account for portion of reads originating from different sequence and paired reads. Not surprisingly, this problem is NP-hard (Kececioglu, 1991).

While accounting for sequencing errors, this formulation is still problematic since it compresses repeated regions in the genome. One way towards solution of this problem is to observe that if we collapse repeated regions, we will see far more reads coming from collapsed regions than from the rest of the genome. This is used in formulation given by Myers (1995). Myers considers a coverage of the output sequence by reads and requires that it is as uniform as possible.

Consider our reconstructed string S, and the positions of reads (layout) consisting of F pairs of integers (s_i, e_i), each pair indicating a starting and ending position of a read in the reconstructed sequence. The layout is ε-valid if for each read A, the edit distance between $S[s_i : e_i]$ and the read is at most $\varepsilon|A|$.

We will now formalize the notion of uniform coverage. Consider an *observed distribution* of read start points (the proportion of reads which start before x):

$$D_{obs}(x) = \frac{|\{s_i < x\}|}{F}$$

We will compare the observed distribution D_{obs} to a distribution of some sampling process D_{src} (also called *source distribution*). Distribution D_{src} is usually uniform, but can be nonuniform due to some systematic errors. We define a maximum deviation between these two distributions as

$$\delta = max|D_{obs}(x) - D_{src}(x)|$$

Definition 4 (DNA sequence reconstruction problem) *Given set of reads $\mathcal{F}$ and error rate ε, find a sequence S and ε-valid layout which has a minimal maximum deviation between observed and source distribution of reads.*

There are no theoretical results known for this formulation (the original paper focuses on developing branch-and-bound algorithm and does not consider NP-hardness or approximation).

The above mentioned formulations have never been used to design a practical sequence assembler, because the exact solutions are too slow and approximate solutions are impractical. A more popular approach is to design heuristic solutions. Most practical assembly algorithms are not backed by a well defined formulation, proof of correctness, or approximation guarantees. In general, these heuristics proceed to "glue" reads which can be unambiguously glued together, building the assembly step-by-step without following clear optimization criteria. They use efficient representation of overlaps between reads and try to resolve ambiguous regions using paired reads and long reads.

A good review of assembly algorithms can be found in Miller et al. (2010). The algorithms can generally be divided into two types, based on the overlap representation.

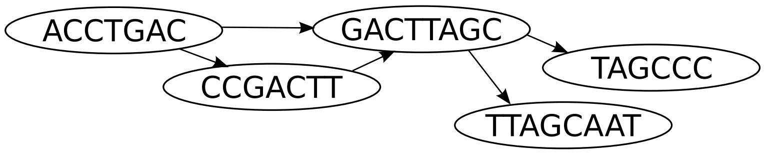

Figure 2.1: **Example of overlap graph for reads `ACCTGAC`, `CCGACTT`, `GACTTACG`, `TAGCCC`, `TTAGCAAT`.**

Overlap-layout-consensus algorithms. The overlap-layout-consensus (OLC) algorithms, first suggested by Myers (1995), work with overlaps between reads, which are usually represented by an overlap graph (see Figure 2.1). In an overlap graph, two reads are connected by an edge if there is a sufficiently long overlap between them.

An OLC assembler works in three major steps. First, we have to find overlaps between reads. We usually allow small edit distance between overlapping parts of reads. Naive implementation of overlapping step has quadratic running time in the number of reads, but it can be speeded up by using various seeding heuristics, where we first look for exact matches of small prespecified length.

After finding overlaps and constructing an overlap graph, we perform the layout step. Major part of the layout step is removal of transitive edges. We remove an edge from u to w if there is an vertex v and edges from u to v and from v to w. After the transitive edges removal, we attempt to find a reasonable layout of reads in the assembly. Each contig from the assembly can be represented as a path in the assembly graph. Assemblers usually start by joining reads which can be joined unambiguously (i.e. when read u is only one following read v and v is only one preceeding u). and then use information from paired reads to resolve the areas around repeats.

Finally, in consensus step we perform basecalling, i.e. if multiple reads overlap in one position and they disagree, we decide the base using majority voting.

A typical example of an OLC algorithm is Celera (Myers et al., 2000). It was mainly used for assembling of Sanger reads. The OLC framework was not suitable for assembling typical Illumina datasets due to a large number of rather short reads (three to ten times shorter than Sanger reads), which would dramatically increase the number of overlaps. The OLC algorithms gained popularity again after introducing PacBio reads which are longer, but have higher error rate. Due to high error rate, the overlap phase is quite challenging, but doable with careful implementation (Myers, 2014).

Nowadays, OLC algorithms are used for assembling long reads with high error rate.

One example is Celera successor Canu (Koren et al., 2017). To handle noisy reads, Canu first corrects reads and then runs the OLC algorithm.

De Bruijn graphs. Another family of algorithms is based on de Bruijn graphs. These algorithms do not work directly with reads, but with sequences of k consecutive bases (k-mers) which occur in reads. The nodes in de Bruijn graph represent k-mers and edges represent adjacencies between k-mers in reads. Note that time required to construct this graph is linear in the length of reads, but on the other hand we lose long range connectivity information since we are not working with complete reads but instead we cut each read to smaller k-mers. After constructing de Bruijn graph we can join k-mers which be unambiguously joined (see more in Section 2.3) and then apply various heuristics for resolving repeats. We will describe de Bruijn graphs in more detail in the next section. A typical example of de Bruijn assembler is Velvet (Zerbino and Birney, 2008).

There are some more technical aspects of the assembly software. Some assemblers like AbySS (Simpson et al., 2009), allow distributed computing of the assembly. Also many assemblers have special requirements for the input data. For example, Allpaths-LG (Gnerre et al., 2011) requires two Illumina read libraries, one with small insert size (apx. 150 bases), and one with longer insert size (apx. 3000 bases), but optionally it can also use information from other libraries (paired reads with very long insert sizes or PacBio reads).

2.3 De Bruijn Graphs

In this section, we discuss de Bruijn graphs, which are an important tool for handling short reads, and serve as a foundation for many assembly algorithms. We will discuss definition and construction of de Bruijn graphs, techniques for handling sequencing errors, and extensions of de Bruijn graphs for handling paired and noisy reads.

A de Bruijn graph (de Bruijn and Erdos, 1946) is a structure for representing all possible overlaps between strings of length k.

Definition 5 (De Bruijn graph) *For given k and alphabet size m, a de Bruijn graph is a directed graph, with m^k vertices, where each vertex represents one string of length k. The edges of de Bruijn graph represent possible overlaps of length $k-1$, i. e. if have have an edge (u, v) and vertex u represents k-mer $a_1, a_2, \ldots, a_k$, then vertex v represents k-mer $a_2, a_3, \ldots, a_k, x$ and k-mer in vertex v can follow k-mer in vertex u.*

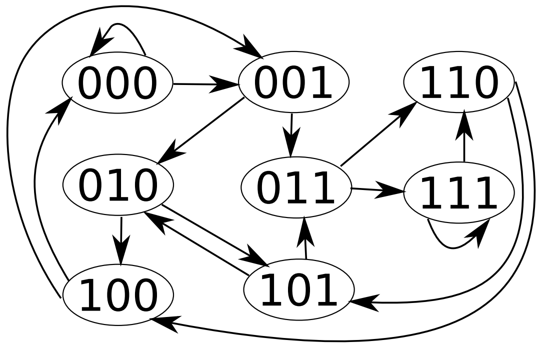

Figure 2.2: **Example of de Bruijn graph for $k = 3$ and alphabet $\{0, 1\}$.**

Figure 2.2 shows an example of de Bruijn graph for $k = 3$ and $m = 2$. Besides application in DNA sequencing, de Bruijn graphs are in many other areas, including design of fault tolerant networks (Baker, 2011) and distributed hash tables (Kaashoek and Karger, 2003).

2.3.1 De Bruijn Graphs for Sequence Assembly

For sequence assembly, de Bruijn graphs were first used by Pevzner et al. (2001). To accommodate specifics of the task, several modifications were introduced to the concept (Pevzner et al., 2001; Zerbino and Birney, 2008).

In particular, to represent reads from both strands, for each vertex v there is a twin vertex v' which represents the reverse complemented k-mer. Union of v and v' is called a block. During manipulation of de Bruijn graphs, any change applied to node v is also symmetrically applied to its twin v'. To ensure that there is no k-mer that would be the same as its reverse complement, k must be an odd number. Example of de Bruijn graph is in Figure 2.3.

Vertices u, v in the de Bruijn graph can be connected by a directed arc if the last $k-1$ bases of u are the same as the first $k-1$ bases in v. Also note that if there is an edge between vertices u, v, there is an inverted edge between their twins, i.e. there is an edge from v' to u'.

We construct de Bruijn graph from reads in a straightforward way. We extract every k-mer from each read and its reverse complement. Then we join by edge each pair of k-mers which are consecutive in some read. Some assemblers also record simple statistics that will be useful in later stages of the assembly process, such as multiplicity of each vertex or edge. Note, that in our application, we are only working with k-mers which are present in some

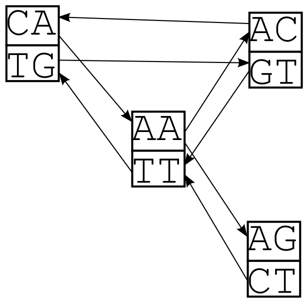

Figure 2.3: **Example of de Bruijn graph constructed from reads: `AACA`, `CTTG`, with $k = 2$.**

reads (compared to the original definition using all possible k-mers).

This construction can be done efficiently by using hash tables, like in Velvet (Zerbino and Birney, 2008). There are also space-efficient representations of de Bruijn graphs based on Bloom filters (Chikhi et al., 2012).

After constructing de Bruijn graph, the assembly can be represented as a set of walks in the graph. Individual assemblers differ in criteria on optimal set of walks and in particular algorithms how to find them efficiently. One approach is to look for an Eulerian path through the graph (Pevzner et al., 2001). Even better approach is to look for an *Eulerian superpath* – the path which visits every read, i.e. it contains sequence of k-mers from each read as contigous subsequence.

Definition 6 (Eulerian superpath problem) *Given graph G and set of paths $\mathbb{P}$, find path P which contains every path from $\mathbb{P}$ as a subpath.*

Euler assembler (Pevzner et al., 2001) solves some instances of Eulerian superpath problem using simple transformations, transforming an Eulerian superpath instance to an equivalent Eulerian path instance.

Other assemblers, like Velvet (Zerbino and Birney, 2008), will join vertices which can be joined unambiguously. More precisely, two vertices u, v can be joined if there is an edge from u to v and that edge is the only outgoing edge from u and the only incoming edge to v. When all possible vertices are joined, each vertex represents one contig of the final assembly.

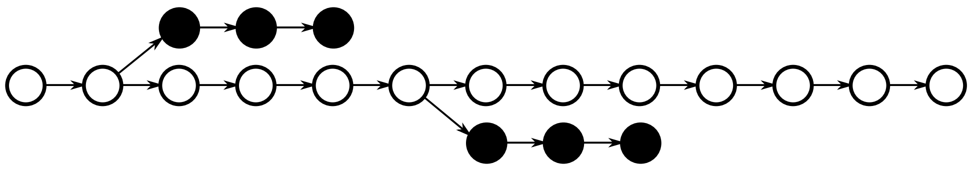

Figure 2.4: **Example of two tips (in black).**

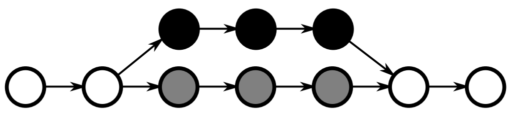

Figure 2.5: **Example of bubble with two paths marked in black and gray.**

2.3.2 Handling Sequencing Errors

Sequencing errors in reads increase complexity of de Bruijn graphs (Pevzner et al., 2001). Common artefacts in a graph that appear due to the sequencing errors are tips and bubbles (see Fig. 2.4, 2.5). Assemblers like Velvet (Zerbino and Birney, 2008), and Abyss (Simpson et al., 2009) handle errors directly during the construction of de Bruijn graph by detecting these artefacts and removing them.

The naive approach would be to remove all k-mers with low coverage. This can be problematic, since some sequencing technologies do not produce reads with uniform coverage, and consequently we would remove good low coverage regions and the graph would become disconnected. Instead, several more sophisticated techniques have been proposed to remove most of the error artefacts from de Bruijn graphs.

Removing tips. A tip is a chain of vertices disconnected at one end. Tips usually represent some local errors in the reads. Their removal is pretty straightforward and does not destroy connectivity in the graph. But sometimes, the tips could represent correct sequences interupted by a gap in the coverage. Velvet (Zerbino and Birney, 2008) removes tips which are at most $2k$ bases long and only if the edge starting the tip has a lower multiplicity than some othe edges in the branching vertex.

Removing bubbles. Bubbles are compose of two or more paths which share the start and end vertices and contain similar sequences (see Fig. 2.5). Bubbles are typically caused by small errors in the middle of the read. Velvet finds such similar paths using the following algorithm: We start a breadth first search-like algorithm from an arbitrary vertex, but consider higher coverage arcs as shorter. Whenever we encounter a previously visited

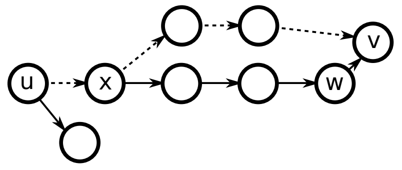

Figure 2.6: **Example of bubble finding.** We start a search from node u. The shortest path between u and v is shown using dashed edges. When searching node w, we will visit node v again. Now we can find the closest common ancestor of current path to w and the shortest path to v, which is vertex x.

vertex v, we try to find the closest common ancestor of the current path and the shortest path to v (see Fig. 2.6). Then we extract the sequences belonging to both paths and if these two paths are judged to be similar enough, the longer path is merged into the shorter one.

Another option is to use tools like QUAKE (Kelley et al., 2010), which try to correct reads without assembling them. These tools usually consider k-mers with low abundance as erroneous and attempt to correct them using a few simple edits in reads.

After correcting for errors, we join vertices, which can be joined unambiguously as described above. We can use this vertices as a base for forming contigs of the assembly. Most assemblers go further and use various heuristics to incorporate information from long and paired reads. In the next section we will look more into handling paired reads.

2.3.3 Handling Paired Reads

De Bruijn graphs do not incorporate any information from paired reads. Usually heuristics are employed to handle paired reads.

ABySS (Simpson et al., 2009) uses paired reads in the following way. After an initial error correction, we align paired reads to reasonably long vertices. Two vertices are considered linked if there are at least p read pairs (usually $p = 5$) which join these two vertices. We denote P_i the set of vertices which are linked to vertex v_i. Then we perform search for single unique path from v_i through the graph which visits each vertex from P_i. Search is limited by using various heuristics (upper bound on number of visited vertices in search and using distance estimates for distances between v_i and elements in P_i). This process is performed for each vertex and consistent paths are stitched to the final assembly.

ALLPATHS-LG assembler (Gnerre et al., 2011) uses a more complicated approach. It requires at least two Illumina short read libraries: one with short insert size (around 100 - 200 bases) and one with medium insert size (around 3000 bases). Their main idea is to find all possible fillings of the gap in the middle of a paired read using using other reads (without regard to their pairing). Then they merge the filled reads to produce the assembly.

To reduce the computational complexity of this approach, they split the assembly problem into several regions and assemble them separately in a process called localization. Localization starts by building de Bruijn graph of the assembly and building initial contigs from it. Then we select long enough contigs as seeds. The region of an assembly is build from the seed, its neighborhood (everything within 10000 bases from the seed) and all reads from the neighborhood. We assemble each region separately and then merge the results into the final assembly.

2.3.4 Variants of de Bruijn graphs

In several works, authors decided to use variants of de Bruijn graphs incorporating additional information besides the k-mers. One important variant are paired de Bruijn graphs (PDGs) (Medvedev et al., 2011). To handle paired reads, PDGs use *k-bimers* as a basic unit of the graph. k-bimer is a pair of two k-mers (a, b), with known distance between them. PDG is constructred and used similarly to de Bruijn graph. This works well, when we know exact distances between reads. If the distance between reads is only approximate, we need to add polishing steps, which merges k-bimer coming from the same location.

It is generally thought, that de Bruijn graphs are not well suited for assembly of noisy reads, since the resulting graph would be highly contaminated by the noise. ABrujin graphs (Lin et al., 2016) solve this problem by constructing the graph only from selected "good" k-mers, which are then connected by labeled edges with distances between connected k-mers.

2.4 Evaluating the Quality of Assemblies

As shown in earlier sections, assembly algorithms are mostly heuristics. Since the assembly usually serves as a basis for other genome analyses (including finding genes, calculating evolutionary histories, etc.), any errors in the assembly process will have a large impact on downstream analyses. Often, we have access to several variants of the assembly and we need to choose the best one. Thus it is important to consider methods for systematically evaluating the quality of assemblies.

In this section, we give an overview of several techniques and statistics for evaluating genome assemblies. We start with simple indicators based only on the assembly. We also describe indicators based on comparing the assembly to the correct answer (the correct answer is usually available in cases when we are benchmarking assemblers). Finally, we describe a probabilistic framework for evaluating the assembly quality, which will form the basis for our own work.

2.4.1 Basic Statistics

Statistics based only on the assembly. To evaluate continuity of the assembly, it is vital to look into distribution of contig lengths. We give a brief overview of metrics used in QUAST (Gurevich et al., 2013), which is a standard tool for assembly evaluation.

First, we consider a total length of contigs. We can have expected assembly length from other sources so if the total length of contigs is far from the expected value, we can already see a problem with the assembly. Number of contigs gives us an approximate notion of fragmentation of the assembly. Sometimes it is better to look at the number of contigs longer than x (where usually $x = 1000$), since very small contigs are usually artefacts of the assembly process. Other statistics for contig length distribution include the length of the largest contig, and Nx (where $0 < x \leq 100$), the largest contig length L, such that using contigs of length $\geq L$ accounts for at least $x\%$ of the bases of the assembly.

Note that each of this statistics can be "gamed", so we should take results based on them cautiously. Problems with them often happen in practice, for example Salzberg et al. (2012) shows in his experiments that some assemblers have higher N50 but also higher number of errors.

Statistics based on the assembly and the reference sequence. If we have access to the true sequence, we can compute various statistics summarizing the number of errors in the assembly. Calculating these statistics usually starts with aligning the assembly to the reference genome, which gives us information in the form: "Substring of the assembly starting at position a and ending at position b can be mapped to the substring of the reference genome starting at c and ending at d with e edits." We call these aligned substrings blocks.

Assembler	No. of contigs	N50 (thousands)	No. of missassemblies	NA50 (thousands)
ABySS	246	34	1	28
ALLPATHS-LG	**12**	**1092**	**0**	**1092**
SGA	456	208	0	208
Velvet	45	762	17	126

Table 2.2: **Comparison of several assemblers on *Staphylococcus aureus* dataset.** Note that some assemblers produce quite high $N50$, but also produce many missassemblies, and their *NA50* is much lower.

After finding aligned blocks, we find *missassemblies*, which are defined as positions in the assembly where block on the left aligns more than 1000 bases away from the block on the right in the reference or the two blocks align to the opposite strands or to different chromosomes (Gurevich et al., 2013).

Now we can count the number of missassemblies and number of contigs containing a missassembly. We can also look for small differences and count the number of substitutions, insertions, or deletions. We can also introduce metrics similar to Nx, called NAx, computed in the same way as Nx, but before computing this statistics, we break contigs at missassemblies.

Table 2.2 show an example of usage of these metrics from GAGE (Salzberg et al., 2012), a project benchmarking the performance of several assemblers.

2.4.2 Probabilistic Models

In many cases, we have to compare multiple assemblies and decide which one is the best. Ghodsi et al. (2013) showed that a very simple and theoretically sound probabilistic model can provide a good indication of the assembly quality.

We will consider a probabilistic model that defines probability $\Pr(R|A)$ that a set of sequencing reads R is observed assuming that the assembly A is the correct assembly of the genome. Since the sequencing itself is a stochastic process, it is very natural to characterize concordance of reads and an assembly by giving a probability of observing a particular read.

Basics of the likelihood model. The model assumes that individual reads are independently sampled, and thus the overall likelihood is the product of likelihoods of the reads: $\Pr(R|A) = \prod_{r\in R} \Pr(r|A)$. To make the resulting value independent of the number of reads in set R, we use as the main assembly score the log average probability of a read computed as follows: $\mathrm{LAP}(A|R) = (1/|R|)\sum_{r\in R} \log \Pr(r|A)$. Note that maximizing $\Pr(R|A)$ is equivalent to maximizing $\mathrm{LAP}(A|R)$.

If the reads were error-free and each position in the genome was sequenced equally likely, the probability of observing read r would simply be $\Pr(r|A) = n_r/(2L)$, where n_r is the number of occurrences of the read as a substring of the assembly A, L is the length of A, and thus $2L$ is the length of the two strands combined.

Another way of looking at the error free model is to say that the probability of generating read from position j is one if read exactly matches the assembly at a given position, and zero otherwise. This can be extended to account for sequencing errors. The probability of generating a read from position j is a real number representing likelihood of generating that read from given position. This value mainly depends on the number of differences between the read and the assembly at the position.

Formally we define $p_{r,j}$ as a probability of generating read r from a sequence that ends at position j. Then the probability of generating read r can be computed as:

$$\Pr(r|A) = \frac{\sum_j p_{r,j}^{forward} + \sum_j p_{r,j}^{reverse}}{2L}$$

The individual probabilities $p_{r,j}$ can be computed by dynamic programing, where we define $T[x, y]$ as a probability of generating prefix of a read of length y from the sequence which ends at position x. Clearly, $p_{r,j} = T[j, \ell]$, where ℓ is the length of the read. Also $T[x, 0] = 1$ for all x and $T[0, y] = 0$ for all $y > 0$. For general x, y, $T[x, y]$ can be computed using following formula:

$$\begin{aligned} T[x,y] = T[x-1,y-1]\Pr(Subs(A[x],r[y])) \\ +T[x,y-1]\Pr(Ins(r[y])) \\ +T[x-1,y]\Pr(Del(A[x])) \end{aligned}$$

Here, $A[x]$ and $r[y]$ represents bases in the assembly at position x and in read at position y respectively. $Subs$, Ins, and Del represent events of substitution, insertion and deletion.

In practice, this dynamic programming is too time consuming. To approximate the probability, we will instead align reads to the assembly and compute likelihood from these alignments.

The probability of a single alignment with m matching positions and s errors (substitutions and indels) is defined as $R(s,m)/(2L)$, where $R(s,m) = \epsilon^s(1-\epsilon)^m$ and ϵ is the sequencing error rate.

Given the read r and set S_r of several best alignments of r to genome A (as obtained by one of standard fast read alignment tools), the probability of generating read r can be estimated as:

$$\Pr(r|A) \approx \frac{\sum_{j\in S_r} R(s_j,m_j)}{2L},$$

where m_j is the number of matches in the j-th alignment, and s_j is the number of mismatches and indels implied by this alignment. The formula assumes the simplest possible error model, where insertions, deletions, and substitutions have the same probability, and ignores GC content bias. Of course, much more comprehensive read models are possible (see e.g. Clark et al. (2013)).

Paired reads. Likelihood model can also accommodate paired reads. We assume that the insert size distribution in a set of reads R can be modeled by a normal distribution with known mean μ and standard deviation σ. The probability of observing paired reads r_1 and r_2 can be estimated from the sets of alignments S_{r_1} and S_{r_2} as follows:

$$\Pr(r_1,r_2|A) \approx \frac{1}{2L}\sum_{j_1\in S_{r_1}}\sum_{j_2\in S_{r_2}} R(s_{j_1},m_{j_1})R(s_{j_2},m_{j_2})\Pr(d(j_1,j_2)|\mu,\sigma)$$

As before, m_{j_i} and s_{j_i} are the numbers of matches and sequencing errors in alignment j_i respectively, and $d(j_1,j_2)$ is the distance between the two alignments as observed in the assembly. If alignments j_1 and j_2 are in two different contigs, or on inconsistent strands, $\Pr(d(j_1,j_2)|\mu,\sigma)$ is zero.

Reads that have no good alignment to A. Some reads or read pairs do not align well to A, and as a result, their probability $\Pr(r|A)$ is very low; our approximation by a set of high-scoring alignments can even yield zero probability if set S_r is empty. Such extremely low probabilities then dominate the log likelihood score. Ghodsi et al. (2013) propose a method that assigns such a read a score approximating the situation when the read would be added as a new contig to the assembly. In practice this usually means having lower bound on the probability of generating a read.

Assembler	LAP	N50 (thousands)	NA50 (thousands)
Reference sequence	−23.509	2873	2873
ALLPATHS-LG	−23.760	1092	1092
SOAPdenovo	−23.862	332	288
Velvet	−23.925	762	126
AbySS	−24.584	34	28

Table 2.3: **Comparison of several assemblers on *Staphylococcus aureus* dataset using LAP and NA50.** Data from Ghodsi et al. (2013).

To illustrate usufulness of likelihood socre we present several results from Ghodsi et al. (2013) where they compare several assemblers and resulting $NA50$ and the assembly likelihood, and show that there is a good correlation between these measures (Table 2.3). More data can be found in Ghodsi et al. (2013).

There are also other approaches to calculate assembly likelihood, most notably ALE (Clark et al., 2013) and CGAL (Rahman and Pachter, 2013). Both models are specifically tailored to paired reads. CGAL model is very similar to the model described above. ALE model consists from three scores, one for read placement, one for insert sizes, and one for read coverage.

2.5 Genome Assembly by Maximum Likelihood

Section 2.2 reviewed several approaches to genome assembly, where most of them are tailored for specific combination of read libraries. Combination of sequencing technologies with complementary strengths can help to improve assembly quality. However, it is not feasible to design new algorithms for every possible combination of datasets. Often it is possible to supplement previously developed tools with additional heuristics for new types of data. For example, PBJelly (English et al., 2012) uses Pacific Biosystems reads solely to aid gap filling in draft assemblies. Assemblers like PacbioToCa (Koren et al., 2012) or Cerulean (Deshpande et al., 2013) use short reads to improve the quality of Pacific Biosystems reads so that they can be used within traditional assemblers. However, such approaches do not use all information contained within the datasets.

We propose a new framework GAML (Genome Assembly by Maximum Likelihood), that allows systematic combination of diverse datasets into a single assembly, without requiring a particular type of data for specific heuristic steps. We build GAML on top of probabilistic model from Section 2.4.2, where we have shown, that probabilistic models are very successful in evaluating the quality of genome assemblers. Here, we use likelihood of a genome assembly as an optimization criterion, with the goal of finding the assembly with the highest likelihood. Even though this may not be always feasible, we demonstrate that optimization based on simulated annealing can be very successful at finding high likelihood genome assemblies.

To test our framework, we have implemented a prototype genome assembler GAML (Genome Assembly by Maximum Likelihood) that can use any combination of insert sizes with Illumina or 454 reads, as well as PacBio reads. The starting point of the assembly are short contigs derived from Velvet (Zerbino and Birney, 2008) with very conservative settings in order to avoid assembly errors. We then use simulated annealing to combine these short contigs into high likelihood assemblies. We compare our assembler to existing tools on benchmark datasets, demonstrating that we can assemble genomes of up to 10 MB long with N50 sizes and error rates comparable to ALLPATHS-LG (Gnerre et al., 2011) or Cerulean (Deshpande et al., 2013). For larger genomes, we can start from an assembly given by a different tool and improve on the result. While ALLPATHS-LG and Cerulean each require a very specific combination of datasets, GAML works on any combination.

2.5.1 Probabilistic Model for Sequence Assembly used in GAML

Probabilistic model used in GAML is build on top of the model by Ghodsi et al. (2013), described in Section 2.4.2. The probabilistic model defines the probability $\Pr(R|A)$ that a set of sequencing reads R is observed assuming that assembly A is the correct assembly of the genome. In our work, instead of evaluating the quality of a single assembly, we use the likelihood as an optimization criterion with the goal of finding high likelihood genome assemblies. We used the following modifications of Ghodsi et al. (2013) model in GAML.

Variable length reads that have no good alignment to A. Ghodsi et al. (2013) propose a method that assigns a read which does not have a good alignment a score approximating the situation when the read would be added as a new contig to the assembly. We modify their formulas for variable read length, and use score $e^{c+k\ell}$ for a single read of length ℓ or $e^{c+k(\ell_1+\ell_2)}$ for a pair of reads of lengths ℓ_1 and ℓ_2. Values k and c are scaling constants set similarly as by Ghodsi et al. (2013). These alternative scores are used instead of the read probability $\Pr(r|A)$ whenever the probability is lower than the score.

Multiple read sets. Our work is specifically targeted at a scenario, where we have multiple read sets obtained from different libraries with different insert lengths or even with different sequencing technologies. We use different model parameters for each set and compute the final score as a weighted combination of log average probabilities for individual read sets $R_1, \ldots, R_k$:

$$\mathrm{LAP}(A|R_1, \ldots, R_k) = w_1\mathrm{LAP}(A|R_1) + \cdots + w_k\mathrm{LAP}(A|R_k). \tag{2.1}$$

In our experiments, we use weight $w_i = 1$ for most datasets, but we lower the weight for Pacific Biosciences reads, because otherwise they dominate the likelihood value due to their longer length. The user can also increase or decrease weights w_i of individual sets based on their reliability.

Penalizing spuriously joined contigs. The model described above does not penalize obvious misassemblies when two contigs are joined together without any evidence in the reads. We have observed that to make the likelihood function applicable as an optimization criterion for the best assembly, we need to introduce a penalty for such spurious connections. We say that a particular base j in the assembly is *connected* with respect to read set R if there is a read which covers base j and starts at least k bases before j, where k is a constant specific to the read set. In this setting, we treat a pair of reads as one long read. If the assembly contains d disconnected bases with respect to R, penalty αd is added to the $\mathrm{LAP}(A|R)$ score (α is a scaling constant).

Computation of likelihood and aligning of data from different sequencing technologies. Our model can be applied to different sequencing technologies by appropriate settings of model parameters (see also Table 2.1). For example, Illumina technology typically produces reads of length 75-150bp with error rate below 1% (Quail et al., 2012). For smaller genomes, we often have a high coverage of Illumina reads. Using paired reads technologies, it is possible to prepare libraries with different insert sizes ranging up to tens of kilobases, which are instrumental in resolving longer repeats (Gnerre et al., 2011). To align these reads to proposed assemblies, we use Bowtie2 (Langmead and Salzberg, 2012). Similarly, we can process reads by the Roche 454 technology, which are characteristic by higher read lengths (hundreds of bases).

Pacific Biosciences technology produces single reads of variable length, with median length reaching several kilobases, but the error rate exceeds 10% (Quail et al., 2012; Deshpande et al., 2013). Their length makes them ideal for resolving ambiguities in assemblies, but the high error rate makes their use challenging. To align these reads, we use BLASR (Chaisson and Tesler, 2012). When we calculate the probability $\Pr(r|A)$, we consider not only the best alignments found by BLASR, but for each BLASR alignment, we also add probabilities of similar alignments in its neighborhood. More specifically, we run a banded version of the forward algorithm by Ghodsi et al. (2013), considering all alignments in a band of size three around a guide alignment produced by BLASR.

2.6 Finding a High Likelihood Assembly

Our goal is to find the highest likelihood assembly directly. Of course, the search space is huge, and the objective function too complex to admit exact methods. Here, we describe an effective optimization routine based on the simulated annealing framework (Eglese, 1990).

Our algorithm for finding the maximum likelihood assembly consists of three main steps: preprocessing, optimization, and postprocessing. In *preprocessing*, we decrease the scale of the problem by creating an assembly graph, where vertices correspond to contigs and edges correspond to possible adjacencies between contigs supported by reads. In order to make the search viable, we will restrict our search to assemblies that can be represented as a set of walks in this graph. Therefore, the assembly graph should be built in a conservative way, where the goal is not to produce long contigs, but rather to avoid errors inside them. In the *optimization step*, we start with an initial assembly (a set of walks in the assembly graph), and iteratively propose changes in order to optimize the assembly likelihood. Finally, *postprocessing* examines the resulting walks and splits some of them into shorter contigs if there are multiple equally likely possibilities of resolving ambiguities. This happens, for example, when the genome contains long repeats that cannot be resolved by any of the datasets. In the rest of this section, we discuss individual steps in more detail.

2.6.1 Optimization by Simulated Annealing

To find a high likelihood assembly, we use an iterative simulated annealing scheme. We start from an initial assembly A_0 in the assembly graph. In each iteration, we randomly choose a *move* that proposes a new assembly A' similar to the current assembly A. The next step depends on the likelihoods of the two assemblies A and A' as follows:

- If $\mathrm{LAP}(A'|R) \geq \mathrm{LAP}(A|R)$, the new assembly A' is accepted and the algorithm continues with the new assembly.

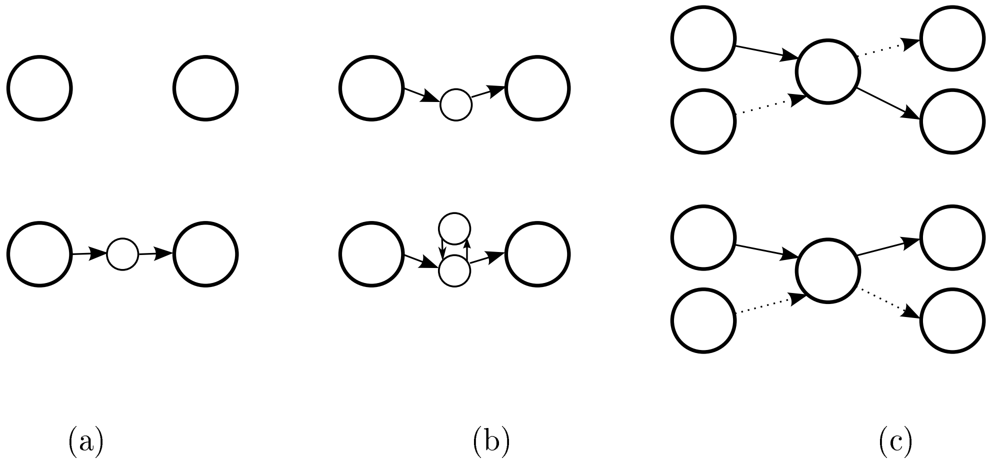

Figure 2.7: **Examples of proposal moves.** (a) Walk extension joining two walks. (b) Local improvement by addition of a new loop. (c) Repeat interchange.

- If $\mathrm{LAP}(A'|R) < \mathrm{LAP}(A|R)$, the new assembly A' is accepted with probability $e^{(\mathrm{LAP}(A'|R)-\mathrm{LAP}(A|R))/T}$; otherwise A' is rejected and the algorithm retains the old assembly A for the next step.

Here, parameter T is called the temperature, and it changes over time. In general, the higher the temperature, the more aggressive moves are permitted. We use a simple cooling schedule, where $T = T_0/\ln(i)$ in the i-th iteration. The computation ends when there is no improvement in the likelihood for a certain number of iterations. We select the assembly with the highest LAP score as the result.

To further reduce the complexity of the assembly problem, we classify all contigs as either *long* (more than 500bp) or *short* and concentrate on ordering the long contigs correctly. The short contigs are used to fill the gaps between the long contigs. Recall that each assembly is a set of walks in the assembly graph. A contig can appear in more than one walk or can be present in a single walk multiple times.

Proposals of new assemblies are created from the current assembly using the following moves:

- *Walk extension.* (Fig.2.7a) We start from one end of an existing walk and randomly walk through the graph, in every step uniformly choosing one of the edges outgoing from the current node. Each time we encounter the end of another walk, the two walks are considered for joining. We randomly (uniformly) decide whether we join the walks, end the current walk without joining, or continue walking.

- *Local improvement.* (Fig.2.7b) We optimize the part of some walk connecting two long contigs s and t. We first sample multiple random walks starting from contig s. In each walk, we only consider nodes from which contig t is reachable. Then we evaluate these random walks and choose the one that increases the likelihood the most. If the gap between contigs s and t is too big, we instead use a greedy strategy where in each step we explore multiple random extensions of the walk of length around 200bp and pick the one with the highest score.

- *Repeat optimization.* We optimize the copy number of short tandem repeats. We do this by removing or adding a loop to some walk. We precompute the list of all short loops (up to five nodes) in the graph and use it for adding loops.

- *Joining with advice.* We join two walks that are spanned by long reads or paired reads with long inserts. We first select a starting walk, align all reads to this walk and randomly choose a read which has the other end outside the walk. Then we find to which node this other end belongs to and join appropriate walks. If possible, we fill the gap between the two walks using the same procedure as in the local improvement move. Otherwise we introduce a gap filled with Ns.

- *Disconnecting.* We remove a path through short contigs connecting two long contigs in the same walk, resulting in two shorter walks.

- *Repeat interchange.* (Fig.2.7c) If a long contig has several incoming and outgoing walks, we optimize the pairing of incoming and outgoing edges. In particular, we evaluate all moves that exchange parts of two walks through this contig. If one of these changes improves the score, we accept it and repeat this step, until the score cannot be improved at this contig.

At the beginning of each annealing step, the type of the move is chosen randomly; each type of move has its own probability. We also choose randomly the contig at which we attempt to apply the move.

Note that some moves (e.g. local improvement) are very general, while other moves (e.g. joining with advice) are targeted at specific types of data. This does not contradict a general nature of our framework; it is possible to add new moves as new types of data emerge, leading to improvement when using specific datasets, while not affecting the performance when such data is unavailable.

2.6.2 Preprocessing and the Initial Assembly

To obtain the assembly graph, we use Velvet with basic error correction and unambiguous concatenation of k-mers. These settings will produce very short contigs, but will also give a much lower error rate than a regular Velvet run. GAML with the default settings then uses each long contig as a separate walk in the starting assembly for the simulated annealing procedure.

2.6.3 Postprocessing

The assembly obtained by the simulated annealing procedure may contain walks with no evidence for a particular configuration of incoming and outgoing edges in the assembly graph. This happens for example if a repeat is longer than the span of the longest paired read. In this case, there would be several versions of the assembly with the same or very similar likelihood score. In the postprocessing step, we therefore apply the repeat interchange move at every possible location of the assembly. If the likelihood change resulting from such a move is negligible, we break the corresponding walks into shorter contigs to avoid assembly errors.

2.6.4 Fast Likelihood Evaluation

The most time consuming step in our algorithm is evaluation of the assembly likelihood, which we perform in each iteration of simulated annealing. This step involves alignment of a large number of reads to the current assembly. However, only a small part of the assembly is changed in each annealing step, which we can use to significantly reduce the running time. Next, we describe three optimizations implemented in our software.

Limiting read alignment to affected regions of the assembly. Since only a small portion of the assembly is affected in each step, we can keep most alignments from the previous iterations and only align reads to the regions that changed. To determine these regions, we split walks into overlapping windows, each window containing several adjacent contigs of a walk. Windows should be as short as possible, but adjacent windows should overlap by at least $2\ell_r$ bases, where ℓ_r is the length of the longest read. As a result, each alignment is completely contained in at least one window even in the presence of extensive indels.

We determine the window boundaries by a simple greedy strategy, which starts at the first contig of a walk, and then extends the window by at least $2\ell_r$ bases beyond the boundary of the first contig. The next window always starts at the latest possible location that ensures a sufficient overlap and extends at least $2\ell_r$ bases beyond the end of the previous window.

For each window, we keep the position and edit distance of all alignments. In each annealing step, we identify which windows of the assembly were changed since the last iteration. We then glue together overlapping windows and align reads against these sequences.

We further improve this heuristics by avoiding repeated alignments of reads to interiors of long contigs, because these parts of the assembly never change. In particular, if some window starts with a long contig, we only realign reads to the last $2\ell_r$ bases from that contig, and similarly we use only the first $2\ell_r$ bases from a long contig at the end of a window.

Reducing the number of reads which need to be aligned. The first improvement eliminates most of the assembly from read mapping. In contrast, the second improvement reduces the set of reads which need to be realigned, because most of the reads will not align to the changed part of the assembly. We use a prefiltering step to find the reads which are likely to align to the target sequence. In the current implementation, we use the following three options for such filtering.

In the simplest approach, we look for reads which contain some k-mer (usually $k = 13$) from the target sequence. We store an index of all k-mers from all reads in a hash map. In each annealing step, we iterate over all k-mers in the target portion of the assembly and retrieve reads that contain them. This approach is very memory consuming, because the identifier of each read is stored for each k-mer from this read.

In the second approach, we save memory using min-hashing (Broder, 1997). Given hash function h, the min-hash of set A is defined as $m(A) = \min_{x \in A} h(x)$. For each read R, we calculate min-hash for the set of all its k-mers. Thus, the identifier of each read is stored in the hash table only once. In each annealing step, we calculate the min-hash for each substring of the target sequence of length ℓ_r and retrieve the reads that have the same min-hash.

An important property of min-hashing is that $\Pr(m(A) = m(B)) = J(A, B)$, where $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ is the Jaccard similarity of two sets A and B (Broder et al., 2000). The statement holds if the hash function h is randomly chosen from a family with the min-wise independence property, which means that for every subset of elements X, each element in X has the same chance to have the minimum hash.

Note that strings with a very small edit distance have a high Jaccard similarity between their k-mer sets, and therefore a high chance that they will hash to the same value using min-hashing. We can use several min-hashes with different hash functions to improve the sensitivity of our filtering at the cost of additional memory.

In our implementation, we use a simple hash function which maps k-mers into 32-bit integers. We first represent the k-mer as an integer (where each base corresponds to two bits). We then xor this integer with a random number. Finally, we perform mixing similar to the finalization of the Murmur hash function (Appleby, 2008):

```
h ^= h >> 16;
h *= 0h85ebca6b;
h ^= h >> 13;
h *= 0hc2b2ae35;
h ^= h >> 16;
```

We choose this finalizer because the Murmur hash function is fast and results in few collisions. It is not min-hash independent, but we found it to perform well in practice.

To illustrate the specificity and sensitivity of min-hashing, we have compared our min-hashing approach with indexing all k-mers (with $k = 15$) on evaluating LAP of the ALLPATHS-LG assembly of *Staphylococus aureus* (using read set SA1 described in Section 2.7 and aligning it to the whole *S. aureus* genome). Indexing all k-mers resulted in 3,659,273 alignments found by examining 21,241,474 candidate positions. Using min-hashing with three hash functions, we were able to find 3,639,625 alignments by examining 3,905,595 candidates positions. Since these reads have a low error rate, k-mer indexing retrieves practically all relevant alignments, while the sensitivity of min-hashing is approximately 99.5%. In min-hashing, 93% of examined positions yield an alignment, whereas specificity of k-mer indexing is only 17%. Also min-hashing used 30 times smaller index.

Note that min-hashing was previously used in a similar context by Berlin et al. Berlin et al. (2014) to find similarities among PacBio reads. However, since PacBio reads have a high error rate, the authors had to use a high number of hash functions, whereas we use only a few hash functions to filter Illumina reads, which have a low error rate.

In GAML, we filter PacBio reads by a completely different approach, which is based on alignments, rather than k-mers. In particular, we take all reasonably long contigs (at least 100 bases) and align them to PacBio reads. Since BLASR can find alignments where a contig and a read overlap by only around 100 bases, we use these alignments as a filter.

Final computation of the likelihood score. When all reads are properly aligned to the new version of the assembly, we can combine the alignments to the final score. In the implementation, we need to handle several issues, such as correctly computing likelihood for reads that align to multiple walks, assigning a special likelihood to reads without any good alignment, and avoiding double counting for reads that align to regions covered by two overlapping windows of the same walk.

Again we improve the running time by considering only reads that were influenced by the most recent change. Between consecutive iterations, we keep all alignments for each sequence window of the assembly and recompute only alignments to affected windows, as outlined above. We also keep the likelihood value of each read or a read pair. Recall that the likelihood of a read or a read pair is the sum of likelihoods of individual alignments.

In each iteration, we then identify which walks were removed and added. Then we calculate likelihoods of all read alignments in these walks (using stored or newly computed alignments) and we use these values to adjust the likelihood values of individual reads, subtracting for removed walks and adding for new walks. At this step, we also handle paired reads, identifying pairs of alignments in correct distance and orientation. Finally, we sum likelihoods of all reads in each dataset and compute the total likelihood score.

2.7 Experimental Evaluation

We have implemented the algorithm proposed in the previous section in a prototype assembler GAML (Genome Assembly by Maximum Likelihood). At this stage, GAML can assemble small genomes (approx. 10 Mbp) in a reasonable amount of time (approximately 4 hours on a single CPU and using 10GB of memory).

To evaluate the quality of our assembler, we have adopted the methodology the GAGE project (Salzberg et al., 2012), using metrics on scaffolds. We have used the same genomes and libraries as Salzberg et al. (2012) (the *Staphylococus aureus* genome and the human chromosome 14) and Deshpande et al. (2013) (the *Escherichia coli* genome). The overview of the datasets is shown in Table 2.4. An additional dataset EC3 (long insert, low coverage) was simulated using the ART software (Huang et al., 2012).

We have evaluated GAML in the following scenarios:

ID	Source	Technology	Insert length	Read length	Coverage	Error rate
Staphylococus aureus (2.87Mbp)						
SA1	Salzberg et al. (2012)	Illumina	180bp	101bp	90	3%
SA2	Salzberg et al. (2012)	Illumina	3500bp	37bp	90	3%
Escherichia coli (4.64Mbp)						
EC1	Deshpande et al. (2013)	Illumina	300bp	151bp	400	0.75%
EC2	Deshpande et al. (2013)	PacBio		4000bp	30	13%
EC3	simulated	Illumina	37,000bp	75bp	0.5	4%
Human chromosome 14 (88.29Mbp)						
H1	Salzberg et al. (2012)	Illumina	150bp	101bp	42	1%
H2	Salzberg et al. (2012)	Illumina	2500bp	101bp	26	3%
H3	Salzberg et al. (2012)	Illumina	35000bp	76bp	1.3	4.5%

Table 2.4: **Properties of datasets used.**

1. combination of fragment and short insert Illumina libraries (SA1, SA2),

2. combination of a fragment Illumina library and a long-read high-error-rate Pacific Biosciences library (EC1, EC2),

3. combination of a fragment Illumina library, a long-read high-error-rate Pacific Biosciences library, and a long jump Illumina library (EC1, EC2, EC3),

In each scenario, we use the short insert Illumina reads (SA1 or EC1) in Velvet with conservative settings to build the initial contigs and assembly graph. For the LAP score, we give all Illumina datasets weight 1 and the PacBio dataset weight 0.01. The results are summarized in Table 2.5. Note that none of the assemblers considered here can effectively run in all three of these scenarios, except for GAML.

In the first scenario, GAML performance ranks third among zero-error assemblers in the N50 length. The best N50 assembly is given by ALLPATHS-LG (Gnerre et al., 2011). A closer inspection of the assemblies indicates that GAML missed several possible joins. One such miss was caused by a 4.5 kbp repeat, while the longest insert size in this dataset is 3.5 kbp. Even though in such cases it is sometimes possible to reconstruct the correct assembly thanks to small differences in the repeated regions, the difference in likelihood between alternative repeat resolutions may be very small. Another missed join was caused

Assembler	Number of scaffolds	Longest scaffold (kb)	Longest scaffold corr. (kb)	N50 (kb)	Err.	N50 corr. (kb)	LAP
(1) *Staphylococus aureus*, read sets SA1, SA2							
GAML	28	1191	1191	514	**0**	514	**−23.45**
ALLPATHS-LG	**12**	1435	**1435**	1092	**0**	**1092**	−25.02
SOAPdenovo	99	518	518	332	**0**	332	−25.03
Velvet	45	958	532	762	17	126	−25.34
Bambus2	17	1426	1426	1084	**0**	1084	−25.73
MSR-CA	17	**2411**	1343	**2414**	3	1022	−26.26
ABySS	246	125	125	34	1	28	−29.43
Cons. Velvet*	219	95	95	31	**0**	31	−30.82
SGA	456	286	286	208	1	208	−31.80
(2) *Escherichia coli*, read sets EC1, EC2							
PacbioToCA	55	1533	1533	**957**	**0**	**957**	**−33.86**
GAML	29	1283	1283	653	**0**	653	−33.91
Cerulean	**21**	**1991**	**1991**	694	**0**	694	−34.18
AHA	54	477	477	213	5	194	−34.52
Cons. Velvet*	383	80	80	21	**0**	21	−36.02
(3) *Escherichia coli*, read sets EC1, EC2, EC3							
GAML	**4**	**4662**	**4661**	**4662**	**3**	**4661**	**−60.38**
Celera	19	4635	2085	4635	19	2085	−61.47
Cons. Velvet*	383	80	80	21	**0**	21	−72.03

*: Velvet with conservative settings used to create the assembly graph in our method.

Table 2.5: **Comparison of assembly accuracy in the first three scenarios.** For all assemblies, N50 values are based on the actual genome size. All misjoins were considered as errors and error-corrected values of N50 and contig sizes were obtained by breaking each contig at each error (Salzberg et al., 2012). All assemblies except for GAML and conservative Velvet were obtained from Salzberg et al. (2012) in the first experiment, and from Deshpande et al. (2013) in the second experiment.

by a sequence coverage gap penalized in our scoring function. Perhaps in both of these cases the manually set constants may have caused GAML to be overly conservative. Otherwise, the GAML assembly is very similar to the one given by ALLPATHS-LG.

In the second scenario, Pacific Biosystems reads were employed instead of jump libraries. These reads pose a significant challenge due to their high error rate, but they are very useful due to their long length. Assemblers such as Cerulean (Deshpande et al., 2013) deploy special algorithms taylored to this technology. GAML, even though not explicitly tuned to handle Pacific Biosystems reads, builds an assembly with N50 size and the number of scaffolds very similar to that of Cerulean. In N50, both programs are outperformed by PacbioToCA (Koren et al., 2012), however, this is again due to a few very long repeats (approx. 5000 bp) in the reference genome which were not resolved by GAML or Cerulean. (Cerulean also aims to be conservative in repeat resolution.) Note that in this case, simulated annealing failed to give the highest likelihood assembly among those that we examined, so perhaps our results can be improved by tuning the likelihood optimization.

The third scenario shows that the assembly quality can be hugely improved by including a long jump library, even if the coverage is really small (we have used 0.5× coverage in this experiment). This requires a flexible genome assembler; in fact, only Celera (Myers et al., 2000) can process this data, but GAML assembly is clearly superior. We have attempted to run also ALLPATHS-LG, but the program could not process this combination of libraries. Compared to the previous scenario, GAML N50 size increased approximately 7 fold (or approx. 4 fold compared to the best N50 from the second scenario assemblies).

2.8 Improving Previously Assembled Genomes

For medium and large genomes, it would take GAML too many iterations to arrive at a reasonable assembly starting from the contigs produced by Velvet with conservative settings. However, it is still possible to scale up GAML to larger genomes by using another assembler to provide a more reasonable starting point.

To this end, we have to map such an input assembly to the assembly graph. We first align the assembly contigs to the Velvet contigs using NUCmer (Delcher et al., 2002). We keep only alignments which cover entire Velvet contigs and have a high sequence identity. If a single input contig is aligned to several Velvet contigs, we connect these Velvet contigs to a walk in the assembly graph. The missing portions of the walk are found by dynamic programming so as to minimize the edit distance between the input contig and the walk.

Assembler	Number of scaffolds	Longest scaffold (kb)	Longest scaffold corr. (kb)	N50 (kb)	Err.	N50 corr. (kb)	LAP
(4) *Human chromosome 14*, starting from Velvet assembly							
Before	1081	4628	263	1190	9156	27	-138.765779
After	1634	1046	265	347	8049	27	-138.632657
REAPR	17727	153	81	36	4607	14	-162.869192
(5) *Human chromosome 14*, starting from ALLPATHS-LG assembly							
Before	129	81640	14918	81640	34	7652	-111.288806
After	139	81640	14918	81640	33	7652	-111.287938
REAPR	858	977	146	190	4230	17	-168.024865

Table 2.6: **Improving existing assemblies of the human chromosome 14 by GAML.** In both experiments, we use read sets H1, H2, and H3 and compare the original assembly computed by another tool with the assembly found by GAML.

In the dynamic programming, we consider only edit distance of up to 10, and if we do not find a connection within this threshold, we add a corresponding number of Ns to our walk.

If the input assembly differs too much from the Velvet contigs, a good mapping of the contigs to walks in the Velvet assembly graph cannot be found. In such cases, we construct the assembly graph directly from the input assembly. We first build a de Bruijn graph from the contigs, and then we concatenate nodes connected by unambiguous connections.

We can now use GAML to improve medium-size genome assemblies (approx. 100 Mbp). In this setting, 10000 iterations require approximately 2 days time and 50GB of memory.

We have tested this approach by using Illumina reads with three different insert sizes (H1, H2, H3) on the human chromosome 14 (data from Salzberg et al. (2012); see Table 2.4). We use the non-conservative Velvet assembly and the ALLPATHS-LG assembly as our starting point. The results are shown in Table 2.6.

Starting from the Velvet assembly, GAML makes 787 breaks and 234 joins, reducing the error count by more than a thousand. Our joins did not introduce any new errors to the assembly. The ALLPATHS-LG assembly has a much higher quality, and starting from this assembly, GAML decreases the number of errors only by one at the cost of introducing ten breaks. In both cases, we were able to remove some assembly errors, while not decreasing the error-corrected N50 values. Perhaps more corrections could be found if we ran our algorithm for more iterations (especially in the Velvet case).

Since breaks predominate in the changes made by GAML, we have also compared our results to REAPR (Hunt et al., 2013), which is a tool that aligns reads to an existing assembly and then splits contigs at the positions weakly supported or even in conflict with the reads. When it concludes that some place is not a breakpoint, but should instead contain an insertion, it inserts a sequence of Ns. Note that REAPR can only process one jumping library along with an optional fragment library, and it requires the library to have a reasonable coverage (15x). Due to these constraints, we have used REAPR only with short jump library H2. For the Velvet assembly, REAPR removes significantly more errors than GAML, but at the cost of a great increase in the number of contigs and a decrease in the error-corrected N50 value. REAPR also introduces many cuts in the ALLPATHS-LG assembly and the GAGE error checking tools report a high increase in errors. We hypothesize that this due to REAPR adding many regions of Ns in the corrected assembly, which leads to a high number of small contigs which GAGE checker cannot align correctly.

2.9 Conclusion

We have presented a new probabilistic approach to genome assembly, maximizing likelihood in a model capturing essential characteristics of individual sequencing technologies. It can be used on any combination of read datasets and can be easily adapted to other technologies arising in the future. We have also adapted our tool to improve existing assemblies after converting a given assembly to a set of walks.

Our work opens several avenues for future research. First, we plan to further improve running time and memory and to allow the use of our tool on larger genomes. Second, the simulated annealing procedure could be improved by optimizing probabilities of individual moves or devising new types of moves. Finally, it would be interesting to explore even more detailed probabilistic models, featuring coverage biases and various sources of experimental error.

Chapter 3

Read Indexing and Related problems

The second generation sequencing technologies (such as Illumina) allow us to investigate DNA and RNA sequences at a previously unseen scale. A single sequencing run can produce vast amounts of sequencing reads of lengths 100–150bp that need to be processed by using efficient data structures and algorithms.

Commonly applied first step in processing Illumina reads is to find (possibly approximate) occurrences of these reads in a reference genome. This task, also called *read mapping*, is often dependent on sophisticated indexes of the reference genome, such as uni-directional or bi-directional FM-index (see e.g. Langmead and Salzberg (2012)). After the reads are aligned to the reference genome, we can answer common queries, such as which or how many reads overlap a particular genomic position. These queries find many applications, including variant calling in population genetic analysis, locating transciption factor binding sites, assessing duplication structure of the genome, differential gene expression, and many more.

However, in many cases the reference genome is unknown or incomplete. In such cases, one would still want to preprocess large collections of reads so that similar queries can be processed efficiently. In particular, Philippe et al. (2011) introduced a problem of *read indexing*, where the task is to build an index which can be queried for all reads that contain a particular k-mer as a substring (maximum k is given beforehand). Their data structure, called Gk-array, is based on efficient indexing of a concatenation of all reads in the collection, and can answer these queries in $O(k \log n + |Q|)$ time, where n is the size of the read collection, and $|Q|$ is the number of reads in the answer. Recently, Välimäki and Rivals (2013) introduced compressed Gk-arrays, which decrease the memory use significantly by using compressed suffix arrays (Grossi et al., 2003). These indexing structures find use in many practical applications involving read clustering, k-mer counting, and similarity search.

In this chapter, we explore problems of indexing either a long string or collection of short strings in such way that we can quickly search for all positions of a short fixed length k-mer (typically with $k \leq 32$). In Section 3.1, we first describe FM-index (Li and Durbin, 2009; Ferragina and Manzini, 2000), then in Section 3.2 we propose MH-index, our alternative data structure for searching for fixed length k-mers in a long string. MH-index is based on the idea of min-hashing, which allows us to build fast index which consumes small amount of memory.

We also investigate problems of indexing collections of short strings, in particular those that are the result of a sequencing experiment. Note, that this problem can be reduced to indexing of long string by concatenating strings in the collection and maintaining the positional information. We first present Gk-arrays (Philippe et al., 2011) and compressed Gk-arrays (Välimäki and Rivals, 2013) in Section 3.3, which follow this basic idea. Then in Section 3.4, we present CR-index, our own data structure for indexing reads, which exploits the fact that the reads are likely substrings of some unknown string, sampled at a high coverage. This assumption allows us to design a data structure that is much more memory efficient than compressed Gk-arrays as we demonstrate in the experimental evaluation.

Work on MH-index has been in part included in a bachelor's thesis of Kuljovský (2016), that I have supervised. CR-index was published at the SPIRE 2015 conference (Boža et al., 2015).

3.1 FM-index

In this section, we briefly describe the FM-index data structure (Li and Durbin, 2009; Ferragina and Manzini, 2000). In recent years, FM-Index has become de facto a standard for string indexing. FM-index is based on suffix arrays (Manber and Myers, 1993), Burrows-Wheeler transform (BWT) and to save memory, it uses succinct data structures like wavelet trees (Grossi et al., 2003).

Definition 7 (Suffix array) *Given string T, suffix array SA of T is lexicographic ordering of suffixes of T, i.e. $SA[i]$ is the index of the i-th lexicographly smallest suffix in T.*

Suffix array can be constructed in linear time (Kärkkäinen and Sanders, 2003). In this array of sorted suffixes, we can search for a substring q of string T by finding suffixes beginning with substring q. More specificaly, the goal of the search is to find an interval (b, e) such that b is the smallest number where q is a prefix of $T[SA[b]..n]$ and e is the largest number where q is a prefix of $T[SA[e]..n]$. After finding the matching interval we can lookup positions in the original string by using values from array SA.

Finding the interval (b, e) can be achieved for example by using the binary search in time $O(|q| \lg n)$, and speeded up by using auxiliary data structures like the LCP (longest common prefix) arrays.

Definition 8 (Longest common prefix (LCP) array) *Given string T and its suffix array SA, the longest common prefix array LCP is defined as: $LCP[1] = -1$ and $LCP[i]$ is the length of the common prefix of suffixes starting at positions $SA[i]$ and $SA[i-1]$.*

The LCP arrays are then combined with range minimum queries and binary search to provide faster search algorithm in suffix arrays. More precisely, denote current inteval of binary search as (b_i, e_i). Denote m_b as length of longest common prefix of $T[SA[b_i]..n]$ and q, and similarly m_e as length of longest common prefix of $T[SA[e_i]..n]$ and q. Denote $LCP(i, j)$ the longest common prefix of suffixes numbered i and j. Denote k_i as the middle point between b_i and e_i. Assume that $m_b \geq m_e$ (other case is handled symmetrically). Now if $LCP(b_i, k_i) > m_b$, then next interval for binary search is (k_i, e_i), since the $(m_b + 1)$-th character of suffixes b_i and k_i is same and different to $(m_b + 1)$-th character in q. If $LCP(b_i, k_i) < m_b$, then next interval for binary search is (b_i, k_i). If $LCP(b_i, k_i) = m_b$, we start comparing from $(m_b + 1)$-th character of suffix k_i and q. We also update values m_b and m_e accordingly.

This algorithm compares each character from q only constant number of times and its complexity is $O(|q| + \lg n)$.

Another option is to use Burrows-Wheeler transform and exploit its internal properties for the substring search.

Definition 9 (Burrows-Wheeler transform) *Given string T, where $T[n] = \$$ (a special end of string character), Burrows-Wheeler transform (BWT) of T is constructed by lexicographically ordering all cyclical rotations of T and then taking the last character from each rotation. Formally, let T_i be the rotation starting at the i-th position of T and let X_i be the index of i-th lexicographically smallest rotation. Then the BWT of T is the sequence $B = T_{X_1}[n]T_{X_2}[n] \ldots T_{X_n}[n]$.*

$\texttt{\$GACACGTA}\underline{\texttt{T}}$

$\texttt{ACACGTAT\$}\underline{\texttt{G}}$

$\texttt{ACGTAT\$GA}\underline{\texttt{C}}$

$\texttt{AT\$GACACG}\underline{\texttt{T}}$

$\texttt{CACGTAT\$G}\underline{\texttt{A}}$

$\texttt{CGTAT\$GAC}\underline{\texttt{A}}$

$\texttt{GACACGTAT}\underline{\texttt{\$}}$

$\texttt{GTAT\$GACA}\underline{\texttt{C}}$

$\texttt{T\$GACACGT}\underline{\texttt{A}}$

$\texttt{TAT\$GACAC}\underline{\texttt{G}}$

Figure 3.1: **Sorted cyclical rotations for string `GACACGTAT`.** Note that we have added a special character \$ at the end of the string. Last charecters of rotations, which are concatenated in the BWT are underlined. The BWT of the original string then becomes `TGCTAA$CAG`.

Using the standard algorithms for construction of suffix arrays we can construct BWT of T in $O(|T|)$ time and space. Also, it is possible to reconstruct the original string T from string B in linear time.

An interesting property of the BWT is *LF mapping*. We denote array of the first letters from rotations $F = T_{X_1}[1]T_{X_2}[1]\dots T_{X_n}[1]$. Note, that array F consists of sorted characters from the string T. Then it holds that if rotation T_i ends with the i-th occurence of letter c in B, then rotation T_{i-1} start with the i-th occurence of letter c in F.

The LF mapping allows us to construct a procedure for finding an occurrence of string q in T as a substring. Denote $C(a)$ as the smallest index i where $T_{X_i}[1] = a$. In other words, $C(a)$ is the total number of characters smaller than a. Also denote $Occ(a, i)$ as the number of occurrences of a in $B[1], \dots, B[i]$. If we know the interval (b, e) for q, then the interval for aq is given as:

$$(C(a) + Occ(a, b-1) + 1, C(a) + Occ(a, e))$$

By starting from an empty string, we can use this formula to calculate interval for any pattern, building it from the end of pattern. Note that the am interval for empty string is $(1, |T|)$.

FM-Index uses compressed representation of structures mentioned above. For the DNA string of size n with alphabet of size 4, the naive representation would need $4n\log_2 n$ bits for array *Occ*, and $n \log_2 n$ bits for array S and $2n$ bits for array B. We also need few bits for array C, but the size of that is only proportional to the alphabet size.

We can save space for array *Occ* by only storing each q-th member of the array and computing other values using array B. Another option is to use wavelet trees (Grossi et al., 2003; Ferragina et al., 2009), to store both array B and array *Occ*. We describe the wavelet trees in Section 3.1.1. The space requiments for wavelet trees are $2n + o(n)$ bits for DNA alphabet.

Also, we can store only each m-th value from array *SA*. Other values can be computed by using array *Occ*. By varying the number m we introduce time-memory tradeoff, since lower m means faster query time, but higher memory requirements. More specifically retrieving one value of *SA* takes $O(m)$ lookups in array *Occ* and we need $\frac{n \lg n}{m}$ bits of memory to store values of *SA*.

In some cases we also want to store inverse suffix array (*ISA*, where $ISA[SA[i]] = i$). Inverse suffix array values can be used for finding position of specific suffix in suffix array and thus finding suffixes which have same character in the beginning. Inverse suffix array values can be sampled similarly as suffix array values.

3.1.1 Rank, Select and Wavelet Trees

Here, we describe how to augment a bit vector with rank and select operations using only a small amount of additional memory. We also describe how to extend these operations for arbitrary sequences using wavelet trees. These operations are essential part of any compressed string indexing data structure. We start by defining rank and select operations over a bit vector.

Definition 10 (Rank) *Given a bit vector $B[1..n]$, $rank(i)$ returns number of ones in $B[1..i]$.*

Definition 11 (Select) *Given a bit vector $B[1..n]$, $select(i)$ returns position of i-th one in B. Thus $rank(select(i)) = i$ and $B[select(i)] = 1$.*

Rank and select operation can be implemented trivially to support $O(1)$ time queries while using $O(n \lg n)$ bits of additional memory. However, there exists a better implementation, where rank and select still work in $O(1)$ time, but the data structure uses only additional $o(n)$ bits of memory (Jacobson, 1988; Clark, 1998).

Jacobson (1988) proposes the following solution for the rank query. We first split array B into blocks of length $b = \lfloor \frac{\lg n}{2} \rfloor$. These blocks are then grouped into superblocks of length $s = s \cdot \lfloor \lg n \rfloor$. For each superblock we calculate rank of the bit preceeding it, thus for i-th superblock we have: $R_s(i) = rank((i-1) \cdot s)$.

For each block, we calculate rank of the bit preceeding the block, but only considering region covered by its superblock. For j-th block contained in i-th superblock we have: $R_b(j) = rank(j \cdot b) - rank(i \cdot s)$ Finally, for each possible sequence of length b and all possible positions we precalculate ranks. We denote this table $R_p(S, k)$, where S is a sequence of length b and k is a position.

Using all three precomputed tables, we can easily answer the rank query in a constant time. Table R_s has $\frac{n}{s}$ elements, each one of them has $\lg n$ bits, so it consumes $O(\frac{n}{s} \lg n) = O(\frac{n}{\lg^2 n} \lg n) = O(\frac{n}{\lg n})$ bits. Table R_b has $\frac{n}{b}$ elements, each one of them has $\lg s$ bits, so it consumes $O(\frac{n}{b} \lg s) = O(\frac{n}{\lg n} \lg(\lg^2 n)) = O(\frac{n \lg \lg n}{\lg n})$ bits. Table R_p has $b2^b$ elements and each one of them has $\lg b$ bits, so it consumes $O(b2^b \lg b) = O(2^{\frac{\lg n}{2}} \lg n \lg \lg n) = O(\sqrt{n} \lg n \lg \lg n)$ bits. Thus, all of these auxiliary tables require $o(n)$ bits of auxiliary memory, to suppor rank queries in $O(1)$ time.

The select query is more complicated. Clark (1998) splits original bit vector into blocks of variable length, each containing $\lg n \lg \lg n$ ones. If the block is sparse, we directly store positions of ones. If the block is dense, we repeat the same procedure, and for second-level dense substrings we use similar lookup table as R_p in rank implementation. Thus using $O(\frac{n}{\lg \lg n}) = o(n)$ additional bits, we can support select queries in $O(1)$ time.

We can generalize rank and select queries for an array over arbitrary alphabet.

Definition 12 *Given a string T of length n over an alphabet Σ of size σ:*

- *$rank_c(i)$ returns the number of occurences of character c in $T[1..i]$,*
- *$select_c(i)$ returns the position of the i-th occurence of character c .*

Again, both queries can be implemented to support $O(1)$ time queries while using $O(\sigma n \lg n)$ bits of auxiliary memory. Another option is to use one bit vector for each character, which lowers the memory use to $\sigma n + o(\sigma n)$ bits.

Perhaps the best structure available for non-binary alphabets to date are wavelet trees (Grossi et al., 2003; Ferragina et al., 2009), which have $O(\lg \sigma)$ query time and use $n \lg \sigma + o(n \lg \sigma)$ bits of space. Wavelet tree is a perfect binary tree, build on alphabet symbols, where each node represent subset of Σ. In particular, the root represents whole alphabet and each leaf represents a distinct alphabet symbol. We denote alphabet associated with tree node v as $\Sigma(v)$. We denote the left child of node v as v_l and the right child as v_r, then:

$$\Sigma(v) = \Sigma(v_l) \cup \Sigma(v_r)$$
$$\Sigma(v_l) \cap \Sigma(v_r) = \emptyset$$

To build a wavelet tree for string T, we associate to each node v a string $T(v)$ which is obtained from T formed by omitting symbols not from $\Sigma(v)$. We do not store $T(v)$ explicitely, we only store a bit vector $B(v)$, such that $B(v)[i] = 1$ if $T(v)[i] \in \Sigma v_r$. By augmenting vectors $B(v)$ with rank and select operations, we can easily answer rank and select queries for general alphabet in $O(\lg \sigma)$ time and using $n \lg \sigma + o(n \lg \sigma)$ bits of memory. We also do not need to store the original string T, since that can be easily recovered from the tree.

3.2 Fast String Matching Using Min-Hashing

In this section, we explore a problem of indexing a long string of length ℓ to quickly search for all locations of a fixed length k-mer in this string. Here we assume, that ℓ is very large and k is relatively small ($k \leq 32$).

Traditionally, this problem is solved in two ways:

- Directly indexing all k-mers (and their positions). All k-mers are stored either in a sorted array or in a hash table. The sorted array is usually more compact and hash table has faster retrieval time. We usually get fast constant query time in case of hash tables (if we assume k fits into the machine word), or logaritmic time in case of sorted arrays. Storing all k-mers takes at least $2nk$ bits of memory (assuming standard four letter DNA alphabet) and storing each position requires $n \lg n$ bits of memory and there is also overhead depending on the hash table implementation.

- Using suffix arrays or FM-index (Li and Durbin, 2009; Ferragina and Manzini, 2000) (see also section 3.1). This approach usually takes much less space than indexing all k-mers. On the other hand, querying is slower. The advantage of FM-index is a possibility to trade query time for memory by adjusting the sampling rate s. Querying using FM-index usually takes $O(k + zs)$ time, where z is the number of hits. Storage space requirements for DNA sequences are approximately $2n + O(n)$ bits for wavelet tree of BWT and $\frac{n \lg n}{s}$ for samples of the suffix array.

Even though all of these solutions have comparable asymptotic time and space complexities, a practical performance differs significantly (see Tables 3.1 and 3.2).

In this section, we propose an alternative index structure, *MH-index* (minhash-index), which is mainly inspired by the use of minimizers in KRAKEN (Wood and Salzberg, 2014) and minhashing used in Canu (Koren et al., 2017). Minimizers were also used for speeding up the de Bruijn graph construction (Li et al., 2013; Chikhi et al., 2014). MH-index does not exhibit good theoretical properties in the worst case, but is very efficient in practice.

In parallel with our work a variant of suffix arrays using minimizers to sample suffixes, called SamSAMi was developed (Grabowski and Raniszewski, 2015). Main advantage of SamSAMi over our work is that SamSAMi works for queries of length k and higher while our approach focuses on queries of length exactly k. On the other hand, SamSAMi has to perform additional binary search step, while MH-index find occurences directly and, in general, is conceptually much simpler.

We start first by definition of a minimizer of a k-mer and then describe how to use the minimizers for indexing k-mers in a string.

Definition 13 (Minimizer of a k-mer) *Given k-mer x, number m, and ordering O, a* minimizer of x *is a substring of x of length m with the minimal value in O. If there are multiple possible minimizers, we select the first one.*

One of the possible orderings is lexicographic, but this leads to undesirable properties, like very skewed distribution of minimizers (Wood and Salzberg, 2014). Instead, we use exclusive-or (XOR) operation with predefined constant c before comparing ordering of k-mers.

In our work, we exploit a crucial property of minimizers that adjacent k-mers are likely to share a common minimizer. First, we find a minimizer for each k-mer in the original string. Then we find a union of these minimizers (see Figure 3.2) and construct index only for the minimizers; for each minimizer value, we record positions of their occurences. We also store the original string in addition to the index.

Querying in MH-index is straighforward. Given query q of length k, we first find the minimizer of the query and then check each position indexed for the minimizer for the query match. Note that we use the position of the minimizer in the original query, thus we only examine a single substring for each minimizer hit.

In fact, checking the minimizer hits may seem like a computational bottleneck. For example, consider a string A^n and a query `AAAAAAAC`, where `AAAA` is the minimizer. In

$$\begin{array}{l}
\texttt{C\underline{AG}TT\underline{AC}CT} \\
\texttt{C\underline{AG}TT} \\
\texttt{\ \underline{AG}TTA} \\
\texttt{\ \ GTT\underline{AC}} \\
\texttt{\ \ \ TT\underline{AC}C} \\
\texttt{\ \ \ \ T\underline{AC}CT}
\end{array}$$

Figure 3.2: **Example of minimizers of k-mer for $k = 5$ and $m = 2$ using lexicographic ordering.** For each k-mer the minimizer is marked. There are only two minimizers for all k-mers in this string.

this string we would have to check almost every position in the original string and all the positions of minimizers are false positives.

In reality, typical DNA sequences do not exhibit this behaviour. Moreover, if k is small, checking the minimizer hit essentially requires only comparison of two 64-bit numbers.

Similarly to FM-index, MH-index exhibits time vs. memory tradeoff. By making the length of the minimizer larger, we use more memory, since k-mers would share less minimizers, but the querying time is faster, since there are less false positive hits from the minimizers.

3.2.1 Finding Minimizers in a String

Grabowski and Raniszewski (2015) discuss several options for finding minimizers in a string. Here, we provide two alternatives that are much simpler and faster than previously considered.

Consider a string T of length n, length of the query k and length of the minimizer m. Our goal is to find minimizers for each substring of length k in T. First, we assume that we can fit each substring of length m into the machine word. Thus listing all substrings of length m can be done in $O(n)$ time and we denote the array of these substrings T_m. Now our goal is to find minimums of intervals $T_m[1..k-m+1], T_m[2..k-m+2]$, etc..

Let $p = k-m+1$. We cut array T_m into windows of length $2p$ where two neighbouring windows overlap by p positions, i.e. we have windows $T_m[1..2p], T_m[p..3p], T_m[2p..4p]$, etc. We use the first window to provide answers for intervals $T_m[1..p], \ldots T_m[p..2p]$. Second window will be used for interals $T_m[p..2p], \ldots, T_m[2p..3p]$, etc.

We process each window using the following algorithm. Given a window $T_m[kp..(k+2)p]$

Index	Memory consumption (MB)	Time per query (μs)
Hash table	426	0.4
Sorted array	72	0.5
FM-index (sampling 1)	15	3.6
FM-index (sampling 4)	5.2	4.0
MH-index ($m = 12$)	4.7	0.5

Table 3.1: **Comparison of indices on *Echerichia coli* dataset (4.6 Mb)**

we split the window into two halves and calculate the suffix minima $S[x] = \min T_m[x..(k+1)p]$ for the first half and prefix minima $P[x] = \min T_m[(k+1)p..x]$ for the second half. We can build these arrays in $O(p)$ time. Note that each answer can be recovered using a combination of suffix and prefix minima in a constant time. Thus processing of one window takes $O(p)$ time.

Since we have $O(\frac{n}{p})$ windows, the whole processing takes $O(n)$ time. Note that we do not need to build whole array T_m at once, we only need to have one window available, thus our processing takes only $O(k)$ additional memory.

Other option is to use double-ended queue (deque). Deque will hold indices for the array T_m. We process elements of T_m in order and in deque we only hold indices of elements, which have potential to become minimizer at some point in the future. Thus, following invariant holds in the deque: If deque consists of elements $a_1, a_2, \ldots, a_z$, then if $i < j$ also $a_i < a_j$ and $T_m[a_i] <= T_m[a_j]$.

During processing of element $T_m[i]$, we are looking for minimum of interval $T_m[i-p+1..i]$. We first remove all elements a_j from the back of the deque where $T_m[a_j] > T_m[i]$, since this elements will never become minimizers. Then we put i into the back of the deque. We also remove all elements from the front of the deque where $a_i < i - p + 1$, since these are elements, which are before our interval of the interest. Now the minimizer of the interval $T_m[i-p+1..i]$ is the first element of the deque.

Since each element was once added to the deque and once removed from the deque whole processing takes $O(n)$ time. Also we consume only $O(k)$ additional memory.

Index	Memory consumption (MB)	Time per query (μs)
Hash table	7217	0.4
Sorted array	1293	0.5
FM-index (sampling 1)	325	3.6
FM-index (sampling 4)	107	4.0
MH-index ($m = 12$)	89	2.5
MH-index ($m = 15$)	100	1.4
MH-index ($m = 19$)	237	1.1

Table 3.2: **Comparison of indices on *Human* chromosome 14 dataset (88 Mb)**

3.2.2 Experiments

We tested the MH-index on two DNA sequences: *Escherichia coli* K12 MG1665 (4.6 Mb long) and *Human* chromosome 14 (88Mb long). We compared indexes based on the hash table, the sorted array, the FM-index with sampling rates 1 and 4, and MH-index with several values of m. We recorded memory consumed by the index and query time for executing million queries of length $k = 30$. We chose half of the queries randomly from original string and the other half of the queries as random strings. Results are sumarized in Tables 3.1 and 3.2.

Experimental results show, that for smaller strings, we can achive small memory footprint comparable with FM-index and very fast query time comparable with hash tables. On larger string we are slightly slower than hash tables, but consume much smaller amount of memory. We are also faster than FM-index using comparable amount of memory. We also believe, than further research and better implemention can speed up MH-index even futher.

3.3 Gk-arrays

In this section, we describe Gk-arrays (Philippe et al., 2011) and compressed Gk-arrays (Välimäki and Rivals, 2013), the data structures for indexing large collections of short reads. Both data structures support the following queries for a fixed length k-mer q:

Q1: In which reads does q occur?

Q2: In how many reads does q occur?

Q3: What are the occurence positions of q in the reads?

Q4: What is the number of occurences of q in the reads?

Q5: In which reads does q occur only once?

Q6: In how many reads does q occur only once?

Q7: What are the occurence positions of q in each read where q occurs only once?

Another interesting feature of Gk-arrays, is ability to also specify the query by read ID and position in the read and improving the query time by using this specification. This feature is useful, for example, for counting overlaps between reads.

Gk-arrays start with concatenating all reads without a delimiter into string T. Then they sort all k-mers from T which do not cross read boundary. They use a few more auxiliary structures similar to LCP arrays to speed up query time. Gk-arrays handle queries Q5-7 by just filtering the answers from respective queries Q1-4. Overall, due to high memory consuption, which is caused by the lack of any compressed data structures, Gk-arrays are not very useful in practice.

Compressed Gk-arrays start by concatenating reads with delimiters into string T. First, they build FM-index over T. Then they use the following auxiliary data structures:

- B_{lcp} is a bit vector, where $B_{lcp}[i] = 1$ if and only if $LCP[i] < k$. This structure is used to find suffix array range for given k-mer from the specified read and the position. First, we use an inverse suffix array to find k-mer position in the suffix array. Then we are looking for range $[s, e]$ such that $LCP[i] \geq k$ for all $s < i \leq e$. This range can be found by using B_{lcp} together with rank and select queries. Note that we can compute the LCP array from the BWT (Beller et al., 2013).

- B_{last} is a bit vector, where $B_{last}[i] = 1$ if and only if k-mer $q = T[SA[i]..SA[i]+k-1]$ starting from position $SA[i]$ is the last occurence of q within the corresponding read. This array, together with rank and select queries, can be used to count the number of reads having specified k-mer.

- B_{once} is a bit vector where for each i, where $B_{last}[i] = 1$, we indicate whether the corresponding k-mer occurs only once in the corresponding read. This vector can be used to speed up answer to queries Q5-7.

Using data structures mentioned above, we can achieve good query times for all queries mentioned above. We summarize space and time complexity of compressed Gk-arrays in table 3.3.

Final index size (bits)		$nH_h \log\log_\sigma n + O(n)$
Query time for	a k-mer	$O(k \log \sigma + polylog(n))$
	a read position	$O(\log\log n)$
Additional time to answer:		
Q1:	In which reads does q occur?	$O(\lvert Q1 \rvert \log\log n)$
Q2:	In how many reads does q occur?	$O(1)$
Q3:	What are the occurence positions of q in the reads?	$O(\lvert Q3 \rvert \log\log n)$
Q4:	What is the number of occurences of q in the reads?	$O(1)$
Q5:	In which reads does q occur only once?	$O(\lvert Q5 \rvert \log\log n)$
Q6:	In how many reads does q occur only once?	$O(1)$
Q7:	What are the occurence positions of q in each read where q occurs only once?	$O(\lvert Q7 \rvert \log\log n)$

Table 3.3: **Summary of space and time complexities for compressed Gk-arrays taken from (Välimäki and Rivals, 2013).** H_h denotes h-th order entropy. $|Qi|$ denotes output size for query type i.

3.4 CR-index

Previous approaches (Philippe et al., 2011; Välimäki and Rivals, 2013) were optimized for collections that consist of randomly generated strings. Yet, in many applications (like our GAML framework proposed in Chapter 2), the collection contains reads that have large overlaps, and there can even be many identical reads. In this section, we propose a new data structure CR-index targeted at read collections that are randomly selected short substrings of a given template, sampled to high coverage, with only a few differences compared to the template (e.g., Illumina reads from a given genome at 50× coverage and 1% error rate).

The main idea of CR-index is to use a *guide superstring* G, which contains all reads r_i from the collection R as substrings. The guide superstring is supplemented by additional structures allowing identification of IDs of all reads that align to a particular position in G. The guide superstring will be generally much shorter than the concatenation of all reads and since representation of this string accounts for most of the memory used in the previous indexing structures, it will be possible to reduce the memory footprint significantly on real data.

It may seem that the genome from which the reads originated may be the ideal guide superstring. At the same time, often we want to use CR-index to support the task of genome assembly from reads, so requiring a guide superstring as a prerequisite may seem somewhat circular. However, we do not require a guide string that would be a plausible interpretation of the read collection. Obtaining a plausible genome assembly means resolving sequence repeats correctly (including correct number of repeats and their organization) and joining as many contigs as possible into larger scaffolds, which is a very difficult task. In contrast, the guide string is only required to satisfy the above technical definition; it is allowed to be very fragmented, and the best guide strings will be over-collapsed, with each repeat included only once. Such guide strings can be easily obtained even through the simplest de Bruijn graph approaches or by simple approximation algorithms for building the shortest superstring.

However, there is a problem with errors contained in the reads. Any read that has been changed compared to the original template will likely not align to the original template and thus the guide string would have to be significantly enlarged to also include all the reads with errors. On the other hand, allowing too many differences between guide string and reads would complicate the querying, since during query time we would have to test all possible differences. As a tradeoff, we decided that the best way is to allow only one difference per each k-mer, which gives following definition of the guide string:

Definition 14 (k-guide superstring) *For a given read collection R and number k, a k-*guide superstring *is a string G such that for each read $r \in R$ there exists a substring of G or a reverse complement of a substring of G, denoted s_r, such that any two differences between s_r and r are located more than k bases apart.*

Note that in this work, we allow only substitutions as differences between r and s_r. The query algorithm is illustrated in Figure 3.3. For a given query k-mer x, we search the guide string for all strings at Hamming distance at most one from x and from the reverse complement of x. This bound on Hamming distance is sufficient, because differences between r and s_r are more than k bases apart, and thus the query will overlap at most one difference between the guide string and the target read. After recovering all potential matching reads, we verify that each of them actually contains the original query x as a substring.

Even though this search algorithm is somewhat complicated due to the relaxed definition of the k-guide superstring, we gain significant improvements in memory. For example on *E. coli* dataset, we were able to construct exact superstring of length 224 Mbp and k-guide string of length 108 Mbp.

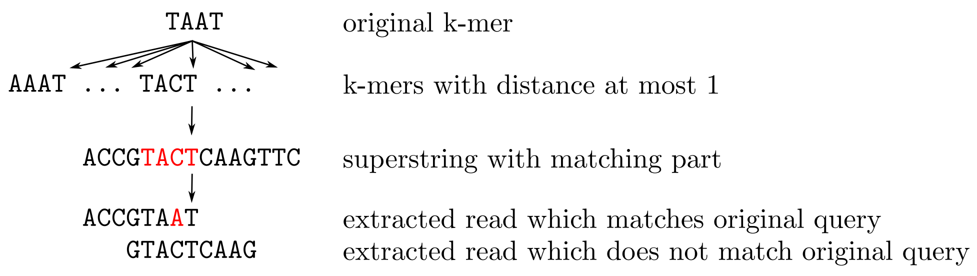
TAAT
original k-mer
AAAT ... TACT ...
k-mers with distance at most 1
ACCGTACTCAAGTTC
superstring with matching part
ACCGTAAT
extracted read which matches original query
GTACTCAAG
extracted read which does not match original query

Figure 3.3: **Overview of the algorithm for answering CR-index query.**

Read 0	`ACCGTAATCA`
Read 1	`   GTACTCAAAG`
Read 2	`      CTGAAAGTTC`
Superstring G	`ACCGTACTCAAAGTTC`
Read positions	0: 0; 1: 3; 2: 6
Read substitutions	(0,6): A, (3,2): G

Figure 3.4: **Overview of the CR-index for three input reads shown at the top.**

3.4.1 Representing k-guide Superstring and Auxiliary Data Structures

As shown in Figure 3.4, the CR-index consists of three main parts. The first part represents the k-guide superstring G and allows fast exact pattern matching. The second part represents the starting positions of individual reads in G and allows us to quickly locate reads covering a given region in G. Finally, the last part lists differences between the reads and the guide superstring, and allows us to quickly verify if a read matches our query k-mer x.

To implement the CR-index, we use succint data structures from the SDSL library (Gog et al., 2014). In particular, the guide superstring is represented using FM-index (Li and Durbin, 2009; Ferragina and Manzini, 2000), which allows efficient exact pattern matching while maintaining a small memory footprint (linear in the length of the superstring). FM-index consists of a wavelet tree, which contains a BWT-transformed superstring G, and samples from suffix and inverse suffix arrays for G. The sampling rate influences memory usage and query time; a higher sampling rate results in faster queries but requires more

Read starts P_b	`1001001000000000`
Read IDs P_r	`0 1 2`
Reverse complements P_c	`F F F`

Figure 3.5: **Data structures needed to locate reads starting at a particular position.**

memory.

The other two parts of the data structure are described in more details below, but both use sparse bit vectors represented as SDarrays (Okanohara and Sadakane, 2007). In this representation, a bit vector of length n with m bits set to one occupies $m \lg \frac{n}{m} + 2m + o(m)$ bits of memory. The rank query (retrieving the number of bits set to 1 in a prefix of the vector of length i) works in $O(\log \frac{n}{m} + \log^4 m/ \log n)$ time and the select query (retrieving the position of i-th bit set to 1) works in $O(\log^4 m/ \log n)$ time.

Representation of reads starting at a given position. After locating a particular k-mer occurrence in the guide superstring G, we need to recover all reads that overlap this occurrence. First, assume that at most one read starts at each position of G. We will construct a bitvector P_b containing 1 at each position where a read starts. This bitvector will be stored in an SDarray, and thus support fast rank and select queries. The read IDs will be stored in array P_r sorted by their position in the superstring. Finally, P_c is a bitvector indicating a strand of the read in the superstring. Figure 3.5 demonstrates the use of these arrays. We can find the read located at position p by first checking whether $P_b[p] = 1$ and then using rank query to find the position of the read in P_r and P_c.

To accomodate multiple reads at the same position, we store reads mapping to the same position in a linked list. Our implementation of the linked lists is optimized for the case when most lists have length only one, which is usually the case unless the coverage is very high. The first item of each linked list is stored in arrays P_b, P_r, and P_c as before. To store the rest of the linked lists, we use a bitvector L_b which contains one at position r if read with ID r has successor in the linked list. Note that the length of L_b is the same as the total number of reads. We enhance this array to support rank queries. Read IDs of the remaining reads (not present in P_r) are in array L_r and their strand information is in bit vector L_c. Figure 3.6 illustrates these structures.

P_b	1010100
P_r	0 3 1
L_b	11000
L_r	4 2

Figure 3.6: **Two reads** `ACG` **with IDs** $0, 4$ **start at position** 0**, one read** GTT **with ID** 3 **starts at position** 2**, and and two reads** TAA **with IDs** $1, 2$ **start at position** 4**.** We omit arrays P_c and L_c for simplicity.

Reads are ordered in L_r and L_c by the ID of their predecessor so that we can use the following algorithm to retrieve all reads starting at position p. We first use arrays P_b and P_r to find index i of the read which is the head of the linked list. The remaining reads are found as follows:

```
while L_b[i] == 1:
  rank = L_b.rank(i)
  output L_r[rank], L_c[rank]
  i = L_r[rank]
```

We store P_r and L_r as ordinary integer vectors with $\lceil \lg(n-1) \rceil + 1$ per integer. In total they take $n(\lceil \lg(n-1) \rceil + 1) + O(\lg n)$ bits. Arrays P_c and L_c are standard bitvectors and take $n + O(\lg n)$ bits in total. Vectors P_b and L_b are sparse, and thus represented as SDarrays.

Differences between the guide and the read. Since not all reads map to the guide string exactly, we need to be able to recover the differences between a particular read and the guide superstring. Let ℓ be the length of one read and n be the number of all reads. Bitvector D_b of length $n\ell$ will for each read and each position in the read store zero, if the read is identical to the guide at that particular position, or one otherwise. Vector D_s will store the differences corresponding to 1s in bitvector D_b. We can use rank and select queries to recover a particular difference. Figure 3.7 illustrates the arrays. We can store the array D_s in $2d + O(\lg d)$ memory, where d is the total number of differences between the reads and the guide. D_b is a sparse bitvector represented by SDarrays.

Bitvector D_b	`0000001000 0000000000 0010000000`
Original bases D_s	`A G`

Figure 3.7: **Representation of differences between the read and the guide.**

Querying CR-index. The algorithm for querying the index follows the outline explained in the beginning of Section 3.4. More specifically, given k-mer x we obtain the reads containing x using the following steps:

1. Construct reverse complement x^R of x and the set Q of all k-mers with Hamming distance of at most one from x or x^R.
2. For each $q \in Q$, find the set P of its positions in G using the FM index.
3. For each $p \in P$, find the set R of reads containing k-mer starting at p. This is achieved by retrieving the reads starting at positions $p - \ell + k, \ldots, p$, where ℓ is the length of a read.
4. For each $r \in R$: If $q = x$ or $q = x^R$ check if r does not contain any substitutions in the interval corresponding to q. Otherwise check if r contains exactly one substitution which is the same as the difference between q and x (x^R). Output the read, if it passes the test.

In the FM index, we search for $O(k)$ string from Q, each search taking time $O(k)$. If this search finds m matching positions in total, we spend $O(m)$ time to recover these positions. Let r be the number of reads overlapping these matches, s the length of G, t the total length of reads, and d the total number of differences. Extracting each read involves a constant number of rank and select queries in arrays P_b and L_b, which takes $O(r \lg \frac{s}{r} + \lg^4 r / \lg s)$ time. Extracting relevant differences takes one rank query in array D_b, which in total takes $O(r \lg \frac{d}{t} + \lg^4 d / \lg t)$.

In our algorithms, we search the data structure for all k-mers from Q. However, if we search for $q \in Q$ which differs from x or x^R by a particular substitution, we may find no matches, because this particular substitution does not occur in any read in the set. In real data, usually only a few substitutions lead to a matching read. To reduce the number of useless queries, we implement a simple filter based on Bloom filters (Bloom, 1970). In particular, we use a Bloom filter to store all k-mers from all reads that differ from their corresponding k-mers in G. For every substituion on a read compared to the superstring we thus store k strings. Before querying FM-index for occurrences of q, we first test whether the substitution in q compared to the query k-mer x or x^R is hashed in the Bloom filter.

3.4.2 Finding the k-guide Superstring

Our goal is to find as short k-guide superstring as possible for a given collection of reads R. This problem is a generalization of a well-known shortest superstring problem, which is NP-hard (Gallant et al., 1980). In our work, we will use a heuristics based on commonly used sequence assembly tools in the following three steps:

1. **Read correction.** First, we identify the reads containing low-frequency substrings. Unique/low-frequency substrings would unnecessarily inflate the size of the guide string, and we attempt to remove these low-frequency substrings by introducing a small number of substitutions. Various formulations of such read correction problem have been studied in the context of sequence assemblers (Kelley et al., 2010). We use the read correction algorithm from SGA (Simpson and Durbin, 2010), but we only accept the substitutions that are at least k bases apart (if SGA proposes substitutions that are too close, we greedily chose a subset satisfying our criterion).

2. **Finding read overlaps.** Again, this is a well studied problem in the context of sequence assembly. We use the SGA overlap algorithm with standard settings.

3. **Construction of the superstring.** The easiest approach would be to use a well-known greedy approximation shortest superstring algorithm, which repeatedly merges two strings with the largest overlap (Blum et al., 1994). In practice, we have found that the guide construction can be speeded up by doing assembly first and only perform the greedy merge as a finishing step to incorporate reads that were not successfuly included in the assembly.

   Various modifications are possible. For example, using a simple string concatenation in the finishing step leads typically to about 25% inflation in the string length compared to the greedy merging.

In our experiments we found that our construction efficiently compressed around 90% of reads (those present in the assembly) and modestly compresses the remaining erroneous reads.

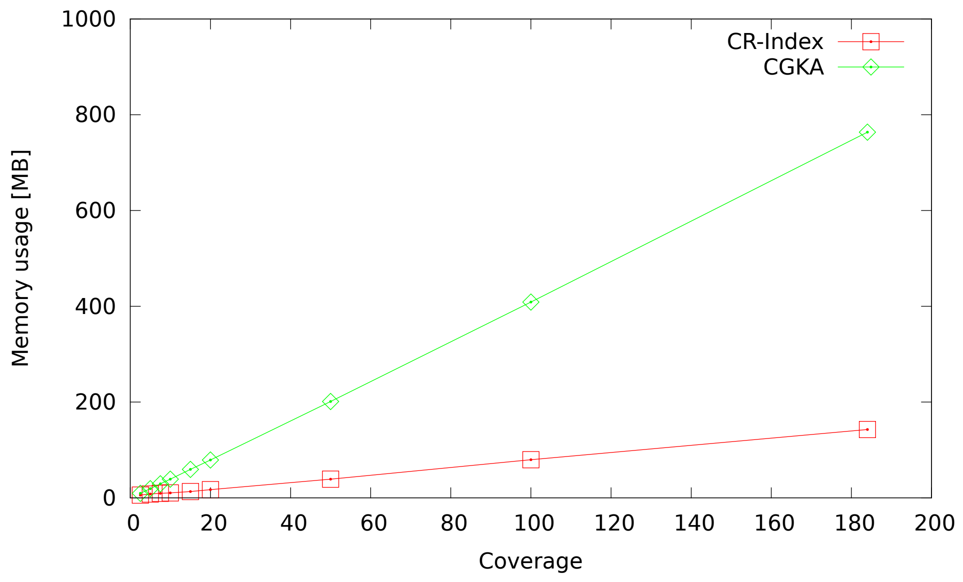

Figure 3.8: **Comparison of memory usage of CR-index and compressed Gk-arrays on the *E. coli* dataset with increasing coverage.**

3.4.3 Experiments

We compare the performance of our data structure with compressed Gk-arrays (Välimäki and Rivals, 2013) on two datasets. The first data set is the set of 151bp Illumina reads from *E. coli* strain MG1655 (genome length 4.7 Mbp, 184× coverage after removal of low-quality reads, 0.75% error rate) (Illumina). The second data set is a set of 101bp Illumina reads from human chromosome 14 (sequence length 107 Mbp, 23× coverage, 1.5% error rate) from Genome Assembly Gold-Standard Evaluations (library 1; Salzberg et al. (2012)). In addition, we also use error-free reads of length 151bp simulated from the *E. coli* genome with coverage up to 50×. In all experiments, we have used query length $k = 15$.

First, we were interested in the guide string length and memory use when dealing with error-free reads from the *E. coli* genome. With increasing number of reads, the size of the guide string stayed almost the same (4.7 Mbp). The overall memory grew linearly with the number of reads, and at 50× coverage, the whole data structure required only 8.8 MB, or 5.9 bytes per read.

The situation is more complicated in the case of real reads containing errors (see Figure 3.8). Memory usage of the CR-index still grows approximately linearly with the coverage, and the slope is much lower than for the compressed Gk-arrays. Whereas the length of the superstring is 6.9 Mbp at coverage 2.5 (1.7 times smaller than total length of reads),

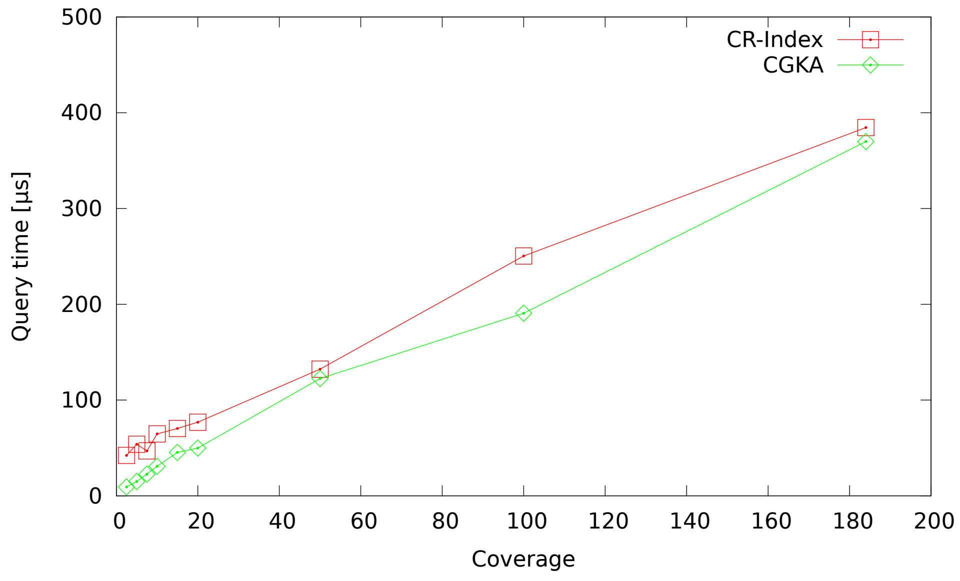

Figure 3.9: **Comparison of query time of CR-index and compressed Gk-arrays on the *E. coli* dataset with increasing coverage.**

the size increases with the coverage up to 108 Mbp at coverage 184, which is 8× smaller than the length of all reads concatenated. The increase in length is due to the presence of reads with errors that are less than k positions apart, and consequently cannot be easily integrated into a smaller superstring. The largest dataset is represented in 142.4 MB of memory, of which 71% is taken by the FM index, 15% is taken by read positions, less than 2% is required to represent the substitutions in reads, and approximately 12% is taken by the Bloom filter.

Compressed Gk-array queries are faster at small coverage, but the relative differences become insignificant at higher coverage (Figure 3.9).

On the human chromosome 14 (23× coverage of 107 Mbp sequence, 1.5% error rate), the CR-index requires only 571 MB of memory, while Gk-arrays require ≈ 1.7 GB. On the other hand, a typical query takes 214 ms in CR-index, or roughly 3× longer than in Gk-arrays (71 ms). We believe that this is due to higher error rate in the data which may cause the Bloom filter to filter out fewer queries.

Both CR-index and compressed Gk-arrays use sampling of the suffix array. This allows time vs. memory tradeoff, since by sampling more values, we get a better running time at the expense of higher memory usage. Figure 3.10 shows this tradeoff at coverage 50 in the *E. coli* dataset. CR-index is much less sensitive to sampling parameters than compressed

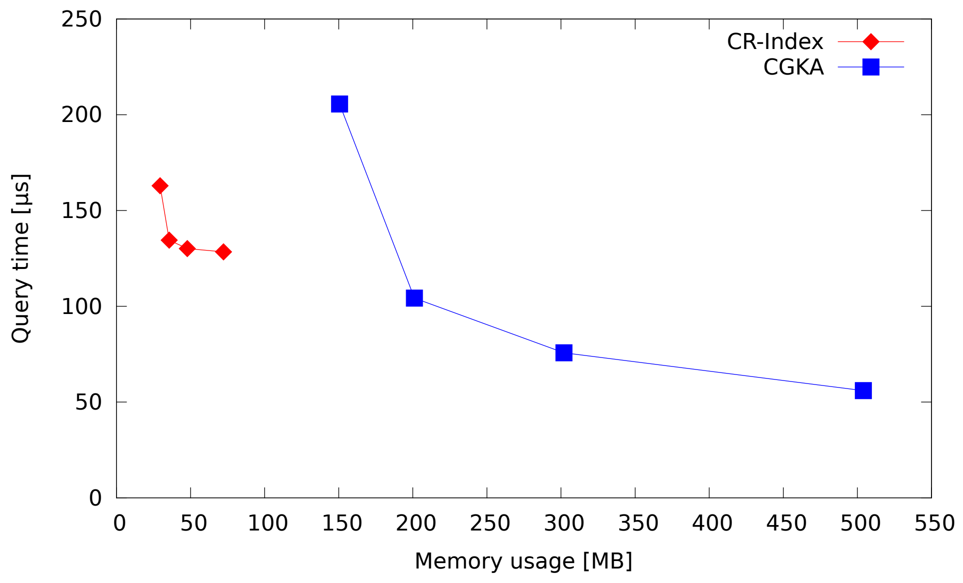

Figure 3.10: **Comparison of query time and memory usage of CR-index and compressed Gk-arrays on the *E. coli* dataset with coverage 50 for varying suffix array sampling rates.**

Gk-arrays. This is due to compressed superstring representation which results in a smaller FM-index and a smaller number of hits in FM-index during the search.

3.4.4 Support of Other Gk-array Queries

CR-index directly supports queries for reads that contain a particular k-mer and for all occurrences of the k-mer in these reads (queries Q1 and Q3 in Välimäki and Rivals (2013)). In this work, we did not implement other queries supported by compressed Gk-arrays. In particular, while the number of reads and the number of occurrences in reads (queries Q2 and Q4) can also be easily recovered, an efficient implementation is more difficult. We could count the reads overlapping a given position in the guide string by rank queries, however these counts will also include reads that mismatch the guide string and these would need to be excluded, requiring additional operations proportional to the number of matching reads. In contrast, after locating the k-mer, compressed Gk-arrays can count the number of occurrences in constant time. Mismatches between the guide string and the reads also do not leave much space for non-trivial implementations of queries Q5-Q7 which require exactly one occurrence of the k-mer in a read. Perhaps additional auxiliary data structures would help to implement these queries in a more reasonable time.

3.5 Discussion and Conclusion

We have presented two data structures for indexing fixed length k-mers either in a long string or in a big collection of short strings.

First, we have presented MH-index, a data structure for indexing long strings, which uses idea of minimzers for lowering its memory complexity. The resulting data structure is conceptually very simple and yet it has a favorable practical performance compared to the hash tables and the FM-index. We think that the MH-index can be improved in case of long repeated character sequences. It might be also worth to combine the idea of minimizers with compressed data structures such as FM-index. Ideally, we would like to have fast average case behaviour of MH-index combined with worse case guaranties of FM-index. Chikhi et al. (2014) uses ordering of minimizers based on their frequency for construction of de Bruijn graph. In future, we should also test effects of this ordering on algorithm performance.

In the second part, we have presented a new compressed data structure, called CR-index, for indexing short sequencing reads. CR-index has a significantly smaller memory footprint than Gk-arrays (Philippe et al., 2011; Välimäki and Rivals, 2013) and still maintains a reasonable query time. In our work, we have used standard algorithms for read correction and sequence assembly to construct the guide string. It may be possible to achive lower memory consumption by developing more specialized algorithms. For error correction, we do not need exact error correction of reads, we need error correction, which minimizes the length of the guide string and thus sometimes correct more errors than needed. Also standard assembly algorithms are quite conservative and we believe, that using aggresive heuristic for constructing shortest common superstring might lead to better results.

In our intended application, which is alignment of multiple references to fixed read library, the same read is likely to be “fished-out” through many k-mer queries, and thus it may be argued that in majority of cases there should be an exact match of at least one k-mer. Thus is most cases in might be possible to skip searching for neighbourhoods to account for mismatches between the guide string and the reads. In this case, we can omit auxiliary structures for error correction and simplify and speed up the query procedure.

Also, in our application we usually query CR-index using k-mers which overlap by $k-1$ positions. Interesting possibility is to support a "rolling" query in a faster time.

Another possible extension of CR-index would be to allow reads to be divided into multiple pieces. The only requirement should be that each k-mer from reads is present in guide string. this idea allows further decrease in memory consumption, which was explored in a followup work by Petrucha (2016).

Chapter 4

Base Calling and Indexing Nanopore Reads

In this chapter, we study problems directly related to DNA sequencing, in particular to MinION nanopore sequencing device. The MinION device by Oxford Nanopore (Mikheyev and Tin, 2014), weighing only 90 grams, is currently the smallest high-throughput DNA sequencer. Thanks to its low capital costs, small size and the possibility of analyzing the data as they are produced, MinION is very promising for clinical applications, such as monitoring infectious disease outbreaks (Judge et al., 2015; Quick et al., 2015, 2016), and characterizing structural variants in cancer (Norris et al., 2016).

Although MinION is able to produce long reads, data produced by the platform exhibit a rather high sequencing error rate.

In this chapter, we first describe data produced by the MinION sequencer, then we examine problems related to base calling of MinION data (translating raw signal into DNA bases). Finally, we explore the area of indexing and aligning raw data from the sequencer. From the computer science perspective, we are mostly looking at problems of sequence to sequence prediction and indexing and comparing sequences of real valued numbers.

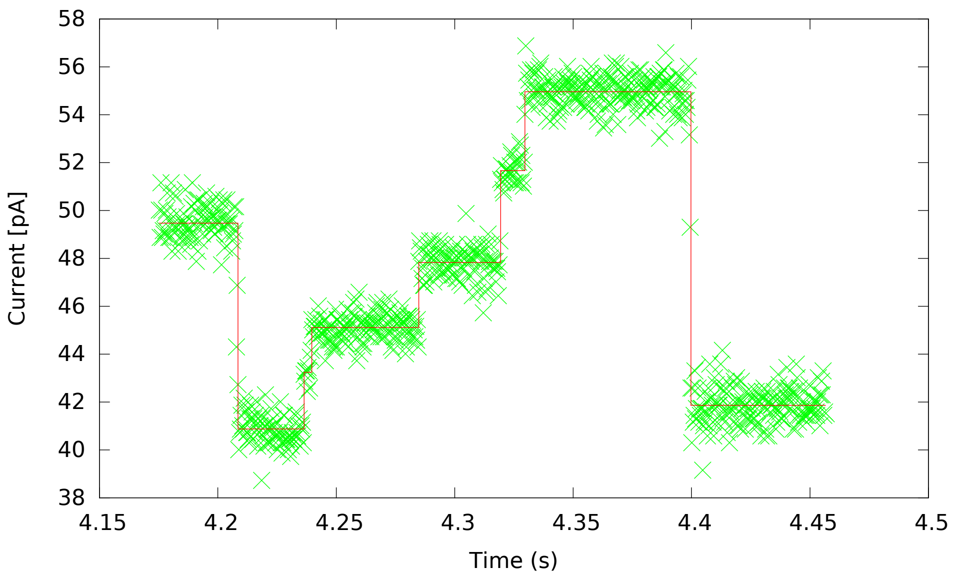

Figure 4.1: **Raw signal from MinION and its segmentation to events.** The plot was generated from the *E. coli* data (`http://www.ebi.ac.uk/ena/data/view/ERR1147230`).

4.1 Characterics of the Data Produced by the MinION Sequencer

In the MinION device, single-stranded DNA fragments move through nanopores. At each pore electric current,affected by the passing DNA fragment, is measured thousands times per second, resulting in a measurement plot as shown in Fig.4.1. The electric current depends mostly on the context of several DNA bases passing through the pore at the time of measurement. As the DNA moves through the pore, the context shifts, and measured signal changes. Based on these changes, the sequence of measurements is split into *events*, each event ideally representing the shift of the context by one base. Each event is summarized by mean, variance and duration. This sequence of events is then translated into a DNA sequence by a base caller.

A MinION device typically yields reads tens thousands bases long; reads as long as 500,000 bp have been reported (Smith, 2016). To reduce error rate, the device attempts to read both strands of the same DNA fragment. The resulting template and complement reads can be combined into a single two-directional (2D) read during base calling. As shown in Table 4.2, this can reduce the error rate of the default base caller from roughly 30% for 1D reads to 13-15% for 2D reads on the R7.3 version.

To sum up, output of the the sequencing device for one read is a sequence of pairs: $(m_1, s_1), (m_2, s_2), \ldots, (m_n, s_n)$, where m_i represents mean of the event i, and s_i represents standard deviation of event i. Optionally, we can have a similar sequence for the complement DNA sequence of the read.

4.1.1 Applications of MinION Sequencer

Apart from obvious applications of MinION data, like genome assembly, where long MinION reads are very helpful, there are many other applications for the MinION sequencer.

Methylation detection. In most of hte current sequencing technologies, the DNA is first multiplied and copied, which results in reading synthetic DNA). In contract to this, MinION can read the original DNA molecules extracted directly from live organism and consequently it can detect modifications in chemical structure of DNA bases, such as methylation (Simpson et al., 2016). These modifications have many regulatory functions in biological organisms. During MinION sequencing, methylation modifications cause small changes in electric current, which can be detected by models specifically trained for this purpose.

Real time sequencing. Another feature of MinION is real time sequencing, where we can obtain results from sequencing as they are produces (even from unfinished reads). This feature is very useful in clinical applications, for example for pathogen detection (Mitsuhashi et al., 2017).

Selective sequencing. Real time sequencing can be further improved with selective sequencing. In many cases, the sample contains a mix of DNA, some of which we are interested in, and some of which is not interesting. For example, when we sequence DNA sample containing DNA from multiple organisms (e.g. human and a pathogen) we are only interested in one organism (e.g. pathogen from the previous example). Or we may be interested only in specific region of DNA.

MinION device has two important features, which allow selective DNA sequencing. First, it allows real time streaming and analysis of sequenced data, and secondly, it is possible to reject particular reads and stop their sequencing. it is thus possible to read only the first few hundred bases from a read (we usually discard the first 100 sequencing events and then look at the next 250 events) from a read, then deciding whether we want

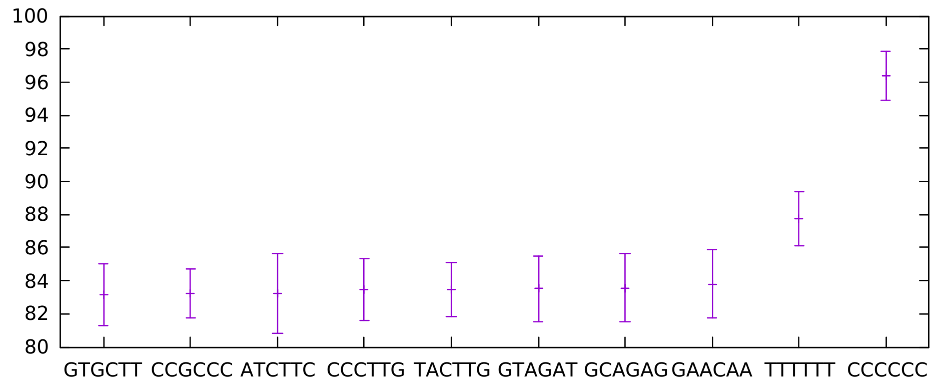

Figure 4.2: **Means and standard deviations of expected signal from MinION for several different kmers.** Data are from Nanocall model (David et al., 2016). We can clearly see that base alling is not a triivial task, since we can get the same signal for multiple contexts.

to keep the read, or stop its sequencing. This idea was mostly explored in Readuntil tool (Loose et al., 2016).

4.1.2 Base Calling of MinION Reads

Since MinION only produces a sequence of real numbers, software is required to translate this sequence of events into the sequence of DNA bases. This software is called base caller. MinION signal sequences are quite noisy (see Fig. 4.2), so we use machine learning approaches for sequence to sequence prediction. Typical examples of these models are Hidden Markov models (HMMs) and recurrent neural networks (RNNs).

The manufacturer of MinION provides proprietary base caller for MinION, called Metrichor. In older versions, the basecaller used Hidden Markov models. In newer version they use recurrent neural networks. When using Hidden Markov models we assume that the electric current read by MinION depends only on a context of $k = 6$ consecutive bases and that the context typically shifts by one base in each step. As a result, every base is read as a part of k consecutive events.

Each state in the HMM model represents one k-tuple and the transitions between states correspond to k-tuples overlapping by $k - 1$ bases (e.g. AACTGT will be connected to ACTGTA, ACTGTC, ACTGTG, and ACTGTT), similarly as in de Bruijn graphs (David et al., 2016). Emission probabilities reflect the current expected for a particular k-tuple, with an appropriate variance added (usually we use a Gaussian distribution for event mean and an Inverse Gaussian distribution for event standard deviation). Finally, additional transitions represent missed events, falsely split events, and other likely errors (in fact, insertion and deletion errors are quite common in the MinION sequencing reads, perhaps due to the errors in the event segmentation).

After the parameter training, base calling can be performed by running the Viterbi algorithm (Jr, 1973), which will result in the sequence of states with the highest likelihood. It is not known, what is the exact nature of the model used in Metrichor, but the emission probabilities required for this type of model are provided by Oxford Nanopore in the files storing the reads. HMM model was also implemented in an opensource basecaller called Nanocall (David et al., 2016).

4.1.3 Signal Scaling and Shift Caused by Pore Degradation

MinION uses biological pores made of proteins. During the sequeicing run these pores degrade, which causes changes in the electric current over time. Fortunatelly, shifts typically do not change significantly within a single read and thus can be modeled by simple scaling and shift of the signal[1]. Detecting correct scaling and shift is very important in HMM models (Nanocall (David et al., 2016) runs a variat of the EM algorithm). In RNN models, it is often sufficient to standardize signal to zero mean and unit variance. We will return to this problem in Section 4.3, where we describe how this problem can affect alignment of raw events.

4.2 Base Calling Using Recurrent Neural Networks

There are several disadvantages to using HMMs for base calling of MinION data. HMMs are very good at representing short-range dependencies (such as moving from one k-mer to the next), yet it has been hypothesized that long-range dependencies may also play a role in MinION base calling, and such dependencies are very hard to capture in HMMs. Also, in HMMs a prior model for the DNA sequence itself is a part of the model. This may be difficult to provide for an unknown DNA sequence and using incorrect prior model may cause significant biases.

In this section, we present our own basecaller DeepNano (Boža et al., 2016), which uses

[1]Some models also assume that signal slightly drifts during sequencing of one read, but effect of drift on accuracy is usually very small.

recurrent neural networks and was developed in parallel with the current official MinION base caller.

4.2.1 Recurrent Neural Network Used by DeepNano

Recurrent neural networks (Lee Giles et al., 1994; Graves, 2012; Goodfellow et al., 2016) are artificial neural networks used for sequence labeling. Given a sequence of input vectors $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_t\}$, the prediction of the recurrent neural network (RNN) is a sequence of output vectors $\{\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_t\}$. In our case, the input vectors consist of the mean, standard deviation, and length of each event, and the output vectors give a probability distribution of called bases.

Simple recurrent neural networks. First, we describe a simple recurrent neural network with one hidden layer (Fig. 4.3a). During processing of each input vector $\vec{x}_i$, a recurrent neural network calculates two vectors: its hidden state $\vec{h}_i$ and the output vector $\vec{y}_i$. Both depend on the current input vector and the previous hidden state: $\vec{h}_i = f(\vec{h}_{i-1}, \vec{x}_i)$, $\vec{y}_i = g(\vec{h}_i)$. We will describe our choice of functions f and g later. The initial state $\vec{h}_0$ is one of the trainable parameters of the model.

Prediction accuracy can be usually improved by using neural networks with several hidden layers (Fig 4.3b), where each layer uses hidden states from the previous layer. We use networks with three or four layers. Calculation for three layers proceeds as follows:

$$\begin{aligned}
\vec{h}_i^{(1)} &= f_1(\vec{h}_{i-1}^{(1)}, \vec{x}_i) \\
\vec{h}_i^{(2)} &= f_2(\vec{h}_{i-1}^{(2)}, \vec{h}_i^{(1)}) \\
\vec{h}_i^{(3)} &= f_3(\vec{h}_{i-1}^{(3)}, \vec{h}_i^{(2)}) \\
\vec{y}_i &= g(\vec{h}_i^{(3)})
\end{aligned}$$

Note that in different layers, we use different functions f_1, f_2, and f_3, where each function has its own set of trainable parameters.

Bidirectional recurrent neural networks. In case of MinION data, the prediction for input vector $\vec{x}_i$ can be influenced by data seen before $\vec{x}_i$ but also by data seen after it. To incorporate this data into prediction, we use a bidirectional neural network (Schuster and Paliwal, 1997), which scans data in both directions and concatenates hidden outputs before proceeding to the next layer (see Fig. 4.3c). Thus, for a three-layer network, the

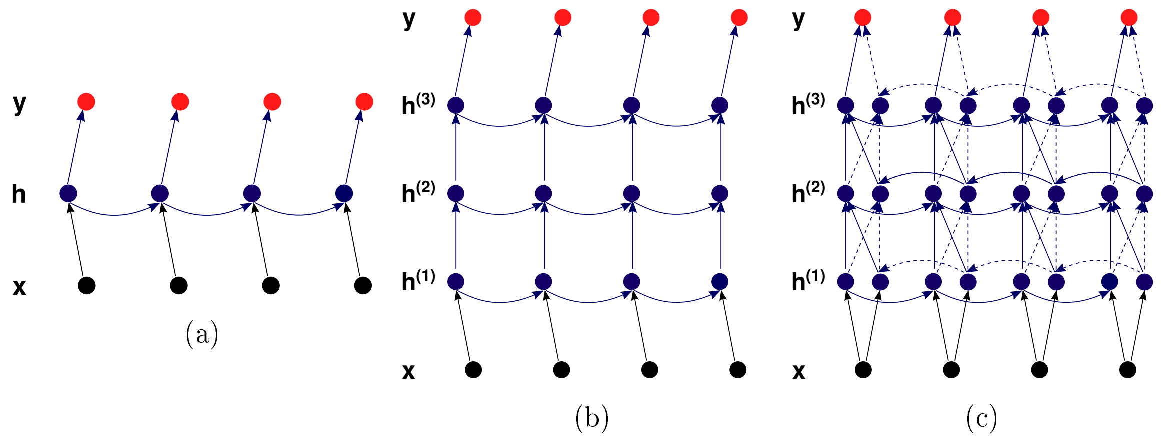

Figure 4.3: **(a) recurrent neural network with one layer. (b) recurrent neural network with three layers. (c) bidirectional recurrent neural network**

calculation would proceed as follows (|| denotes concatenation of vectors):

$$\begin{aligned}
\vec{h}_i^{(1+)} &= f_{1+}(\vec{h}_{i-1}^{(1+)}, \vec{x}_i) \\
\vec{h}_i^{(1-)} &= f_{1-}(\vec{h}_{i+1}^{(1-)}, \vec{x}_i) \\
\vec{h}_i^{(1)} &= \vec{h}_i^{(1+)} || \vec{h}_i^{(1-)} \\
\vec{h}_i^{(2+)} &= f_{2+}(\vec{h}_{i-1}^{(2+)}, \vec{h}_i^{(1)}) \\
\vec{h}_i^{(2-)} &= f_{2-}(\vec{h}_{i+1}^{(2-)}, \vec{h}_i^{(1)}) \\
\vec{h}_i^{(2)} &= \vec{h}_i^{(2+)} || \vec{h}_i^{(2-)} \\
\vec{h}_i^{(3+)} &= f_{3+}(\vec{h}_{i-1}^{(3+)}, \vec{h}_i^{(2)}) \\
\vec{h}_i^{(3-)} &= f_{3-}(\vec{h}_{i+1}^{(3-)}, \vec{h}_i^{(2)}) \\
\vec{h}_i^{(3)} &= \vec{h}_i^{(3+)} || \vec{h}_i^{(3-)} \\
\vec{y}_i &= g(\vec{h}_i^{(3)})
\end{aligned}$$

Gated recurrent units. A typical choice of function f in a recurrent neural network is a linear transformation of inputs followed by hyperbolic tangent nonlinearity:

$$f(\vec{h}_{i-1}, \vec{x}_i) = \tanh(W\vec{x}_i + U\vec{h}_{i-1} + \vec{b})$$

where the matrices W, U and the bias vector $\vec{b}$ are the trainable parameters of the model. Note that we use separate parameters for each layer and direction of the network.

This choice unfortunately leads to the vanishing gradient problem (Hochreiter, 1998). During parameter training, the gradient of the error function in layers further from the output is much smaller that in layers closer to the output. In other words, gradient diminishes during backpropagation through network, complicating parameter training.

One solution is to use long-short term memory units (Hochreiter and Schmidhuber, 1997). In our model we use gated recurrent units (Chung et al., 2014), which are simpler than LSTM and also solve vanishing gradient problem.

Given input $\vec{x_i}$ and previous hidden state $\vec{h}_{i-1}$, a gated recurrent unit first calculate values for update and reset gates:

$$\vec{u_i} = \sigma(W_u\vec{x_i} + U_u\vec{h}_{i-1} + b_u),$$

$$\vec{r_i} = \sigma(W_r\vec{x_i} + U_r\vec{h}_{i-1} + b_r),$$

where σ is the sigmoid function: $\sigma(z) = 1/(1+e^{-z})$. Then the unit computes a potential new value

$$\vec{n_i} = \tanh(W\vec{x_i} + \vec{r_i} \circ U\vec{h}_{i-1}).$$

Here, $\circ$ is the element-wise vector product. If some component of the reset gate vector is close to 0, it decreases the impact of the previous state.

Finally, the overall output is a linear combination of $\vec{n_i}$ and $\vec{h}_{i-1}$, weighted by the update gate vector $\vec{u_i}$:

$$\vec{h}_i = \vec{u_i} \circ \vec{h}_{i-1} + (1 - \vec{u_i}) \circ \vec{n_i}.$$

Note that both gates give values from interval $(0, 1)$ and allow for a better flow of the gradient through the network, making training easier.

Output layer. Typically, one input event leads to one called base. But sometimes we get multiple events for one base, so there is no output for some events. Conversely, some events are lost, and we need to call multiple bases for one event. We limit the latter case to two bases per event. For each event, we output two probability distributions over the alphabet $\Sigma = \{A, C, G, T, -\}$, where the dash means no base. We will denote the two bases predicted for input event $\vec{x_i}$ as $b_i^{(1)}$ and $b_i^{(2)}$. Probability of each base $q \in \Sigma$ is calculated from the hidden states in the last layer using the softmax function:

$$P[b_i^{(k)} = q] = \frac{\exp(\vec{\theta}_q^{(k)}\vec{h}_i^{(3)})}{\sum_{p\in\Sigma}\exp(\vec{\theta}_p^{(k)}\vec{h}_i^{(3)})}$$

Final basecalling is done by taking the most probable base for each $b_i^{(k)}$ (or no base if dash is the most probable character from Σ). During training, if there is one base per event, we always set $b_i^{(1)}$ to dash.

During our experiments we found that outputing two independent distributions works better than outputing one distribution with 21 symbols (nothing, 4 options for one base, 16 options for two bases).

Training. Let us first consider the scenario in which we know the correct DNA bases for each input event. The goal of the training is then to find parameters of the network that maximize the log likelihood of the correct outputs. In particular, if $o_1^{(1)}, o_1^{(2)}, o_2^{(1)}, \dots, o_n^{(1)}, o_n^{(2)}$ is the correct sequence of output bases, we try to maximize the sum

$$\sum_{i=1}^{n} \lg P[b_i^{(1)} = o_1^{(1)}] + \lg P[b_i^{(2)} = o_2^{(2)}]$$

As an optimization algorithm, we use stochastic gradient descent (SGD) combined with Nesterov momentum (Sutskever et al., 2013) to increase the convergence rate. For 2D basecalling, we first use SGD with Nesterov momentum, and after several iterations we switch to L-BFGS (Liu and Nocedal, 1989). Our experience suggests that SGD is better at avoiding bad local optima in the initial phases of training, while L-BFGS seems to be faster during the final fine-tuning.

Unfortunately, we do not know the correct output sequence; more specifically, we only know the region of the reference sequence where the read is aligned, but we do not know the exact pairs of output bases for individual events. We solve this problem in an EM-like fashion. First, we create an approximate alignment between the events and the reference sequence using a simple heuristic (in R7.3 we minimize the sum of differences between the expected and observed means for events and have simple penalties for undetected and split events; in R9 we use outputs of metrichor as approximate targets). After each hundredth pass through the whole data set, we realign the events to the reference sequence. We score the alignment by computing the log-likelihood of bases aligned to each event in the probability distribution produced by the current version of the network. Each event can get zero, one or two bases aligned to it. Score of an alignment for each event is sum of log of output probabalities (either for aligned bases or for empty bases). To find the alignment with maximum likelihood, we use a simple dynamic programming.

Data preprocessing. Mean and standard deviation of measured signals change over time due to degradation of pores during a sequencing run. The simplest way of accounting for this factor is to scale the measured values.

In our model, we can use scaling parameters calculated by Metrichor. In particular, we use scale and shift for mean, and scaling for standard deviation; we do not use drift for means as the use of this parameter has a negligible effect on the performance. To make our approach independent of Metrichor, we have also implemented a simple method for computing scaling parameters. In particular, we set the scaling parameters so that the 25th and 75th percentile of the mean values fit predefined values and the median of the standard deviations fits a predefined value. Using either Metrichor scaling parameters or our simplified scaling yields a very similar performance in our experiments.

1D base calling. The neural networks described above can be used for base calling template and complement strands in a straightforward way. Note that we need a separate model for each strand, since they have different properties. In both models, we use neural networks with three hidden layers and 100 hidden units.

2D base calling. In 2D base calling, we need to combine information from separate event sequences for the template and complement strands. A simple option is to apply neural networks for each strand separately, producing two sequences of output probability distributions. Then we can align these two sequences of distributions by dynamic programming and produce the DNA sequence with maximum likelihood.

However, this approach leads to unsatisfactory results in our models, with the same or slightly worse accuracy than the original Metrichor basecaller. We believe that this phenomenon occurs because our models output independent probabilities for each base, while the Metrichor basecaller allows dependencies between adjacent basecalls.

To compensate for this shortcoming, we have built a neural network which gets as an input corresponding events from the two strands and combines them into a single prediction. To do so, we need an alignment of the two event sequences, as some events can be falsely split or missing in one of the strands. We can use either the alignment obtained from the base call files produced by Metrichor or our own alignment, computed by a simple dynamic programming over output probabilities, which finds path with the highest likelihood. We convert each pair of aligned events to a single input vector. Events present in only one strand are completed to a full input vector by special values. This

	E. coli training	*E. coli* testing	*K. pneumoniae* testing
# of template reads	3,803	3,942	13,631
# of template events	26,403,434	26,860,314	70,827,021
# of complement reads	3,820	3,507	13,734
# of complement events	24,047,571	23,202,959	67,330,241
# of 2D reads	10,278	9,292	14,550
# of 2D events	84,070,837	75,998,235	93,571,823

Table 4.1: **Sizes of experimental data sets.** The sizes differ between strands because only base calls mapping to the reference were used. Note that the counts of 2D events are based on the size of the alignment.

input sequence is then used in a neural network with four hidden layers and 250 hidden units in each layer; we needed to use a bigger network than in 1D case since there is more information to process.

Implementation details. We have implemented our network using Theano library for Python (Bergstra et al., 2010), which includes symbolic differentiation and other useful features for training neural networks. We do not use any regularization, as with the size of our dataset we saw almost no overfitting.

4.2.2 Experimental Evaluation of DeepNano

We have used existing data sets from *Escherichia coli* (Loman et al., 2015a) and *Klebsiella pneumoniae* (WTC Human Genetics, 2016) produced by the SQK-MAP006 sequencing protocol with R7.3 flow cells. We have only used the reads that passed the original base calling process and had a full 2D base call. We have also omitted reads that did not map to the reference sequence (mapping was done separately for 2D base calls and separately for base calls from individual strands).

We have split the *E. coli* data set into training and testing portions; the training set contains the reads mapping to the first 2.5 Mbp of the genome. We have tested the predictors on reads which mapped to the rest of the *E. coli* genome and on reads from *K. pneumoniae.* Basic statistics of the two data sets are shown in Table 4.1.

	E. coli	*K. pneumoniae*
Template reads		
Metrichor	71.3%	68.1%
Nanocall	68.3%	67.5%
DeepNano	77.9%	76.3%
Complement reads		
Metrichor	71.4%	69.5%
Nanocall	68.5%	68.4%
DeepNano	76.4%	75.7%
2D reads		
Metrichor	86.8%	84.8%
DeepNano	88.5%	86.7%

Table 4.2: **Accuracy of base callers on two testing data sets for R7.3 MinION data.** The results of base calling were aligned to the reference using BWA-MEM (Li, 2013). The accuracy was computed as the number of matches in the alignment divided by the length of the alignment.

Accuracy comparison. We have compared our base calling accuracy with the accuracy of the original Metrichor base caller and with Nanocall. The main experimental results are summarized in Table 4.2. We see that in the 1D case, our base caller is significantly better on both strands and in both data sets. In 2D base calling, our accuracy is still slightly higher than Metrichor's.

On the *Klebsiella pneumoniae* data set, we have observed a difference in the GC content bias between different programs. This genome has GC content of 57.5%. DeepNano has underestimated the GC content on average by 1%, whereas the Metrichor base caller underestimated it by 2%.

To explore sequence biases in more detail, we also compared the abundance of all 6-mers in the *Klebsiella* genome in the base-called reads. Fig. 4.4 shows that base calls produced by DeepNano exhibits significantly smaller bias in 6-mer composition than Metrichor base calls. This trend is particularly pronounced for repetitive 6-mers (Fig. 4.5); similar bias was previously observed by Loman et al. (2015b).

With R9 version of the platform, Oxford Nanopore has introduced a variety of base calling options, including cloud-based Metrichor service, local base calling options, ex-

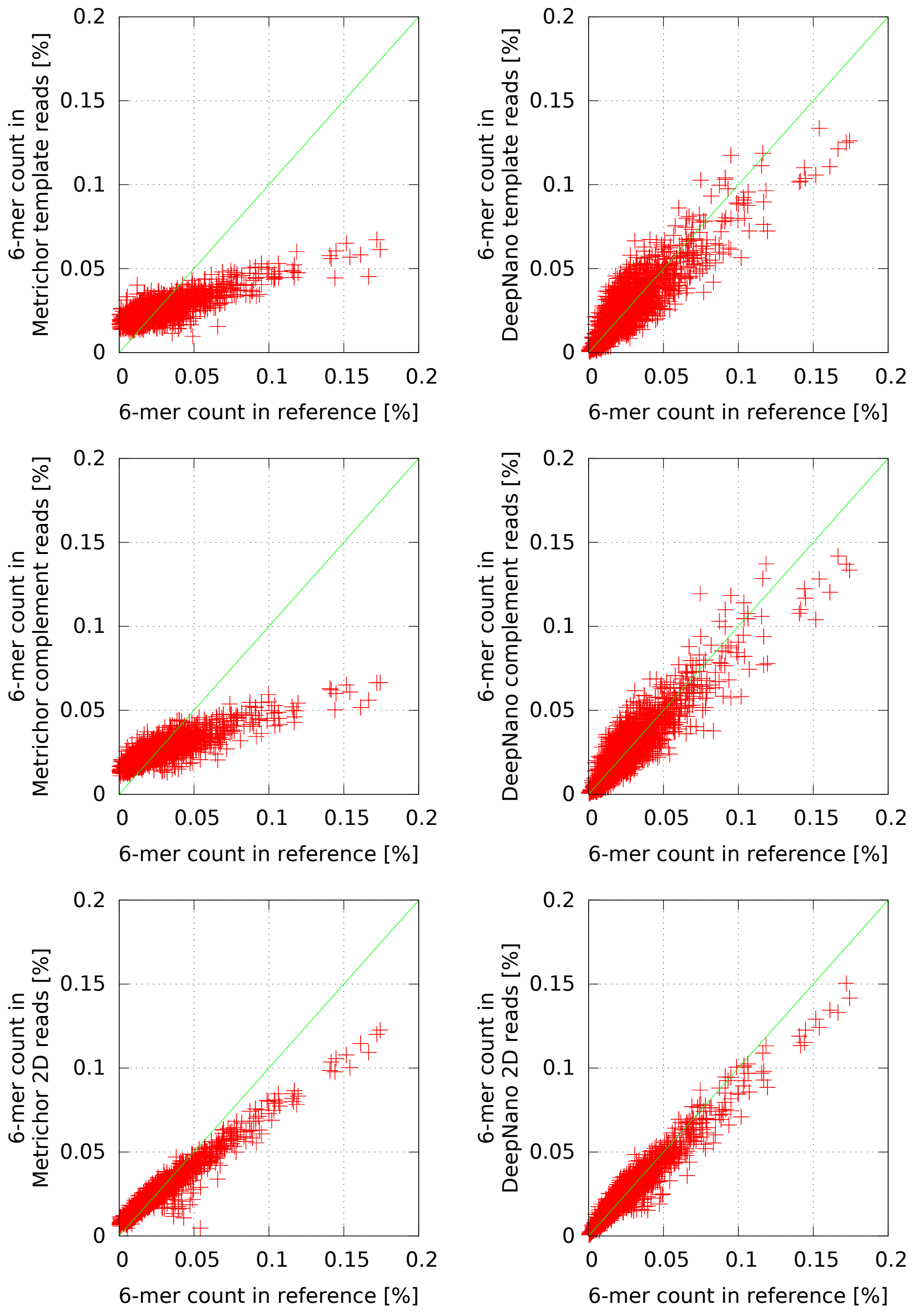

Figure 4.4: **DeepNano reduces bias in 6-mer composition.** Comparison of 6-mer content in *Klebsiella* reference genome and base-called reads by Metrichor (left) and DeepNano (right). From top to bottom: template, complement, 2D.

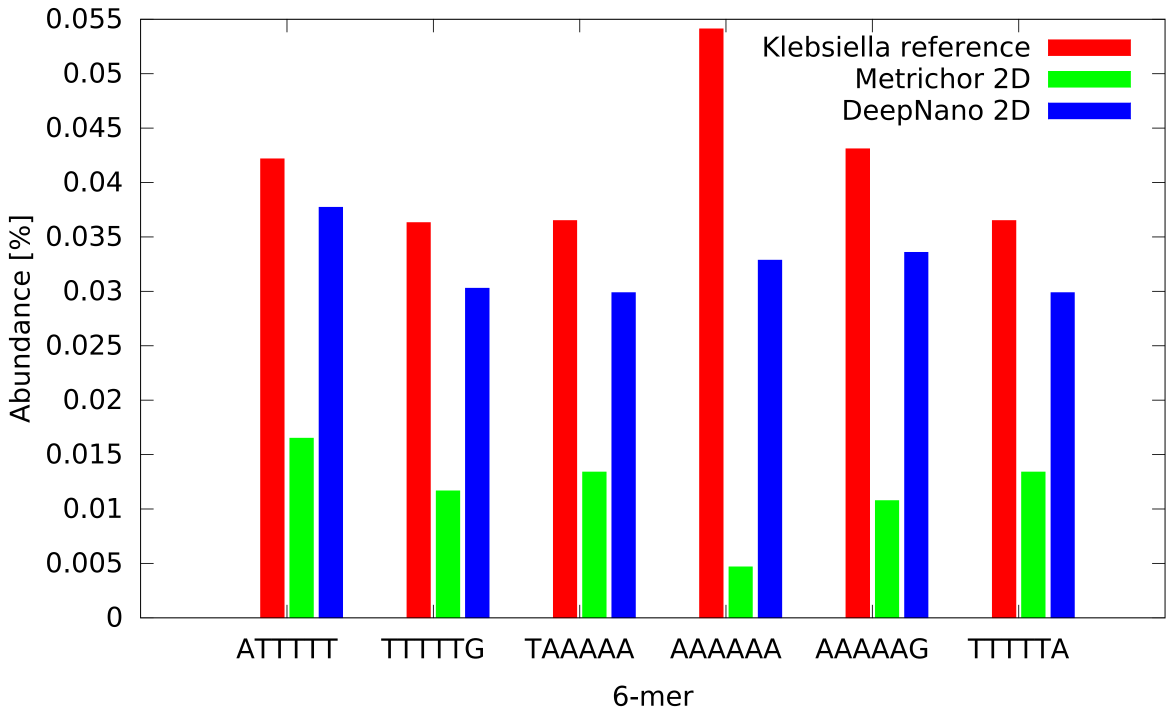

Figure 4.5: **Abudances for repetitive 6-mers.**

perimental Nanonet code base, and binary-only Albacore platform, all of these options very similar in accuracy. All of these options are also based on RNNs. We have used a benchmark *E. coli* data set from Loman lab (`http://s3.climb.ac.uk/nanopore/R9_Ecoli_K12_MG1655_lambda_MinKNOW_0.51.1.62.tar`) to evaluate performance of DeepNano. The training and testing sets were split in the same way as in the case of R7 data sets. Table 4.3 shows that the accuracy of DeepNano is very similar to that of Nanonet, but DeepNano is faster than Nanonet. By decreasing the number of hidden units from 100 to 50, we can further trade accuracy for base calling speed. The smaller RNNs can be used in applications, where fast running times are crucial.

Base calling speed. It is hard to compare the speed of the Metrichor base caller with our base caller, since Metrichor is a cloud-based service and we do not know the exact configuration of hardware used. From the logs, we are able to ascertain that Metrichor spends approximately 0.01 seconds per event during 1D base calling. DeepNano spends 0.0003 seconds per event on our server, using one CPU thread. During 2D base call, Metrichor spends 0.02 seconds per event (either template or complement), while our base

	Accuracy	**Speed (events per second)**
Nanonet	83.2%	2057 ev/s
DeepNano (100 hidden units)	81.0%	4716 ev/s
DeepNano (50 hidden units)	78.5%	7142 ev/s

Table 4.3: **Accuracy and running time on R9 data.** The results of base calling were aligned to the reference using BWA-MEM (Li, 2013). The accuracy was computed as the number of matches in the alignment divided by the length of the alignment.

caller spends 0.0008 seconds per event. To put these numbers into perspective, base calling a read with 4,962 template and 4,344 complement events takes Metrichor 46s for template, 34s for complement, and 190s for 2D data. DeepNano can process the same read in 1.5s for template, 1.3s for complement, and 11.3 seconds for 2D data. We believe that unless Metrichor base calling is done on a highly overloaded server, our base caller has a much superior speed. Compared to Nanocall, we observed that DeepNano is 5 to 20 times faster, depending on Nanocall settings.

Although DeepNano is relatively fast in base calling, it requires extensive computation during training. The 1D networks were trained for three weeks on one CPU (with a small layer size, there was little benefit from parallelism). The 2D network was trained for three weeks on a GPU, followed by two weeks of training on a 24-CPU server, as L-BFGS performed better using multiple CPUs. Note however that once we train the model for a particular version of MinION chemistry, we can use the same parameters to base call all data sets produced by the same chemistry, as our experiments indicate that the same parameters work well for different genomes.

4.3 Fast Indexing and Alignment of MinION Data

One of the MinION applications mentioned in Section 4.1.1 is selective sequencing, where we only sequence "interesting" reads by rejecting all other reads after reading the first few hundred bases. The idea was mostly explored in Readuntil tool (Loose et al., 2016).

Hard part of selective sequencing is deciding which reads to reject. Standard algorithm for selective sequencing would be to base call the first few hundreds of events from the read and then align them to the reference sequence. After aligning, we would accept the read if we find a match with the acceptable similarity.

Unfortunately, the speed of the base calling algorithm on a reasonable computer is much lower than the rate at which the sequencer produces the data. This means that we need to use other approaches than base calling followed by alignment. We will explore an approach which does not translate electric signal into DNA, but instead it works directly with the electric signal data.

First, we describe dynamic time warping (DTW) algorithm used by Loose et al. (2016) and then we demonstrate that using DTW and application of thresholding on its matching cost does not have enough discriminative power to distinguish false matches from true matches. We hypothesize that the main problem lies in determining correct scaling and shift of the data (which is hard on small number of events). We test this experimentaly by using scaling and shift parameters determined from the whole read and our experiments agree with the hypothesis.

Finally, we propose a variant of DTW algorithm and demonstrate that it has much better discriminative power than the original DTW algorithm.

4.3.1 Dynamic Time Warping

Using emission distributions from the HMM for base calling (see Section 4.1.2), we can construct the expected electric signal for given sequence of DNA bases. Thus, our main goal is to compare two sequences of electric signals, characterized by event means. We will ignore event standard deviation in this section. More formally, given an expected sequence $e_1, e_2, \ldots, e_n$ and an observed sequence $o_1, o_2, \ldots, o_m$, we need to determine whether the observed sequence fits into the expected sequence, and if so, find the alignment.

Since the event detection is unreliable, the expected and observed sequences may not only exhibit small differences in signal levels, but also some events may be missing or duplicated. A standard algorithm for comparing two such signal sequences, which vary in speed is called Dynamic Time Warping (DTW) (Sankoff and Kruskal, 1983). DTW was first used for aligning speeches.

DTW calculates a matching between elements of two sequences. It produces a sequence of pairs $(a_1, b_1), \ldots, (a_k, b_k)$, where a_i represents an index in the first signal sequence and b_i represents an index in the second signal sequence. The following properties should hold:

- $b_1 = 1$, $b_k = m$
- $0 \leq b_i - b_{i-1} \leq 1$

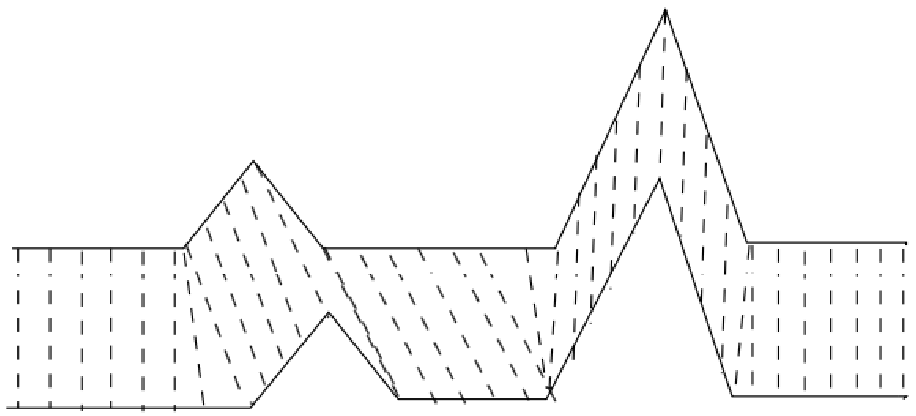

Figure 4.6: **An ilustration of DTW matching between elements.** Note that sometimes a single element of the first sequence is matched to multiple elements from the second sequence (and vice versa), but each element is matched to at least one element from the other sequence.

- $0 \leq a_i - a_{i-1} \leq 1$
- $(a_i > a_{i-1}) \vee (b_i > b_{i-1})$

The goal of DTW is to minimize the cost of the matching, which is the sum of costs for each matched pair. The cost of matching a pair is defined by cost function $d(x, y)$ which compares two matched elements from the two sequences. In our case, we have chosen simply $d(x, y) = (x - y)^2$.

The optimal matching can be found by using a simple dynamic programming. We define subproblem $D[i][j]$ as the best cost of DTW for sequences $e_1, \ldots, e_i$ and $o_1, \ldots, o_j$, assuming that e_i and o_j are matched together. Using this subproblem, the dynamic programming would proceed according to the following recurrence:

$$D[i][j] = d(e_i, o_j) + \min(D[i-1][j-1], D[i-1][j], D[i][j-1])$$

In our application, we are checking whether the second sequence can be found in the first sequence, thus the cost of the optimal alignment can be found as: $\min_i D[i][m]$.

The algorithm has $O(nm)$ time complexity, where n, m are the lengths of the first and second sequence.

4.3.2 Applying DTW to Selective Sequencing

Before running DTW on MinION data, we first have to determine correct scaling and shift of the data. Standard procedure for this is Z-score normalization, i.e. normalizing data mean to zero and variance to one, as used in Readuntil (Loose et al., 2016).

After finding the best matching, we have to decide based on the score, whether the read matches the target sequence or whether the score only represents a spurious match of the query to the target sequence (a false positive). This is usually done by simply thresholding on the score. We will call this setup a *naive scaling*.

In the following sections, we demostrate that the naive scaling has very low specificity when higher sensitivity is required. This causes problems especially when we are trying to match data to a longer reference sequence, where many false positive matches can occur.

Here, we propose a simple heuristics, which greatly improves the tradeoff between sensitivity and specificity. If the alignment of the query was fixed, we could easily find better scaling parameters by looking at the sequence of matched pairs and setting scaling and shift to minimize the matching cost. More formally, given a DTW alignment $(a_1, b_1), \ldots, (a_k, b_k)$, we are looking for parameters A, B such that the matching cost

$$\sum_k (Ao_{b_i} + B - e_{a_i})^2$$

is as small as possible. This can be solved by the ordinary least squares regression, which means calculating:

$$\hat{o} = \frac{1}{k}\sum_i o_{b_i}$$

$$\hat{e} = \frac{1}{k}\sum_i e_{a_i}$$

$$A = \frac{\sum_i (o_{b_i} - \hat{o})(e_{a_i} - \hat{e})}{\sum_i (o_{b_i} - \hat{o})^2}$$

$$B = \hat{e} - A\hat{o}$$

Once we have better scaling parameters, it is possible to improve the matching by running the DTW algorithm on newly rescaled signal. In fact, we could run several iterations of rescaling and DTW to improve the matching and the scaling parameters. We call this approach *iterative scaling*.

In the next section, we compare the performance of the naive scaling and our new iterative rescaling method.

Algorithm variant	Sensitivity	Specificity
Baseline	90%	99.72%
Naive	90%	99.88%
Simple rescaling	90%	**99.94**%
Two iteration rescaling	90%	99.92%
Baseline	95%	99.64%
Naive	95%	99.45%
Simple rescaling	95%	**99.88**%
Two iteration rescaling	95%	99.86%
Baseline	98%	99.46%
Naive	98%	90.74%
Simple rescaling	98%	**99.58**%
Two iteration rescaling	98%	99.46%
Baseline	99%	98.88%
Naive	99%	47.30%
Simple rescaling	99%	95.88%
Two iteration rescaling	99%	**97.20**%

Table 4.4: Comparison of specificity of several scaling variants on prespecified sensitivity levels.

4.3.3 Experimental Evaluation

In our experiments we have used R9 Ecoli dataset from Loman Labs (Loman, 2016) using only reads with and alignment to the reference and longer than 3000 bases.

As a query, we have used events number 100 to 350 from each read. In each experiment, we run each method for each read against aligned location and against hundred random 500 bp long locations from the genome. After running each method, we study specificity-sensitivity tradeoff of each method by varying the threshold for the score. For each method, we set the threshold to achieve a predefined sensitivity (how many times DTW accepted a match of the read to the correct location) and measure specificity (how many times DTW rejected a match of the read to a random location).

We compared the following methods:

- Z-score normalization using complete reads (on more than 3000 events), followed by DTW (on 250 events). This is an unrealistic scenario in which we effectively know the correct scaling parameters. We will use this method as a *baseline* for comparison.

- Naive scaling, which represents the state of the art, consisting of Z-score normalization, followed by DTW.
- Simple rescaling, consisting of Z-score normalization, followed by DTW and rescaling.
- Two iteration rescaling, consisting of Z-score normalization, followed by DTW, rescaling, and another DTW.

Table 4.4 shows comparison of these scaling variants.

We can see that at higher sensitivities, the naive variant performs much worse than rescaled variants of DTW. This is especially visible at 99% sensitivity level, where naive variant treats every second false match as a good match. Also at 99% sensitivity we see benefit of two iteration rescaling, which discards many more false positives than simple rescaling. A consistently strong baseline performance suggests that the scaling is a big problem when using DTW and need to be addressed before the method can be widely applicable. Our approach seems like a step in a good direction to solve the problem.

4.4 Conclusions and Future Work

In this chapter, we have proposed a new base caller DeepNano for MinION data, and we also propose a variant of the DTW algorithm for matching MinION raw data to reference sequence, having much higher specificity at high sensitivity levels than current solutions.

DeepNano provided more accurate base calling on older versions of MinION sequencing platform. For current version, it provides faster but slightly less accurate base calling compared to the base caller provided by the device manufacturer. We think, that the current role of DeepNano is not to compete with the official base caller in accuracy, but to provide a foundation for solving other niche problems, like finding DNA methylation or providing better base calling in cases where other base callers fail (these cases are frequently reported from many laboratories).

There are also many avenues for improving base calling accuracy like, using ensembles of multiple models, using raw unsegmented data, and adjusting architecture of the network. Also, the speed of the base calling can be improved by a technique called dark knowledge (Hinton et al., 2014, 2015) where we train a smaller network using data generated from output probabilities of a larger network. This approach often leads to an improved accuracy for the smaller networks compared to training it directly on the training data. Another approach to speed up the network, is to use quasi-recurrent neural networks (Bradbury et al., 2016), which architecture is much more parallelizable and thus possibly faster than ordinary recurrent neural networks.

We also plan to further improve raw signal indexing. Our goal is to develop a data structure for indexing raw signals, where we could first preprocess target reference sequence and then find candidate positions for possible signal matches for a query sequence, using this index similarly as in classical sequence aligners (Li, 2013; Chaisson and Tesler, 2012; Altschul et al., 1990; Kent, 2002).

Also, an interesting topic to study theoretically is a variation of DTW, where we allow the algorithm to select best scaling and shift parameters.

Chapter 5

Conclusion

In this thesis, we have presented several algorithms related to sequence assembly.

In Chapter 2, we presented GAML assembly framework, which can seamlessly combine multiple types of reads and has good theoretical foundation. There are many possibilities for further research here. First, we believe that optimization procedure can be improved to spend less time on obvious improvements (we can make obvious improvements to assembly in parallel). Secondly, we believe that GAML needs more testing with even more types of sequencing data (optical maps (Nagarajan et al., 2008), RNAseq (Wang et al., 2009), etc.). Finally, the probabilistic model used in GAML can be improved to accomodate more realistic model of errors, like chimeric reads, biases of particular technologies, etc.

In Chapter 3, we have presented two data structures for indexing either a long string or a collection of short strings in such a way that we can quickly search for all positions of a short fixed-length query. MH-index is an index for the long strings, which uses idea of minimizers. While MH-index works in practice, little is known about theoretical properties of methods based on minhashes. It would be interesting to study theoretical properties, such as expected number of minimizers, etc. Also we think, that there is an interesting possibility of combining the idea of minimizers with FM-indices to gain faster search times in practice and still good theoretical guarantees in the worst case. We also presented CR-index, a structure for querying collections of short reads. One open area for research, is to directly support alignment of indexed reads to predefined sequence.

In Chapter 4, we studied problems related to Oxford Nanopore sequencer MinION. First, we have developed our own base caller, DeepNano. It would be interesting to study problems related to DNA methylation, either using supervised or unsupervised learning. Also developing a fast and reasonably accurate base caller to be used in a field sequencing might be possible avenue for DeepNano. Lastly, we studied problems of aligning raw signal data from the sequencer. Here, we demonstrated that naive scaling approaches combined with DTW fail and proposed a better approach. Here, the obvious avenue of research, is to develop an index for raw signal data, to improve the query time.

Bibliography

Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

Austin Appleby. Murmurhash. `https://code.google.com/p/smhasher/wiki/MurmurHash`, 2008.

Joel Baker. De bruijn graphs and their applications to fault tolerant networks. Master's thesis, Oregon State University, 2011. 1, 2011.

Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the burrows–wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, pages 3–10, 2010.

Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality sensitive hashing. *bioRxiv*, page 008003, 2014.

Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM (JACM)*, 41(4):630–647, 1994.

Vladimır Boža, Brona Brejová, and Tomáš Vinař. Gaml: Genome assembly by maximum likelihood. In *Algorithms in Bioinformatics (WABI)*, volume 8701 of *Lecture Notes in Bioinformatics*, page 122. Springer, 2014.

Vladimír Boža, Jakub Jursa, Broňa Brejová, and Tomáš Vinař. Fishing in read collections: Memory efficient indexing for sequence assembly. In *International Symposium on String Processing and Information Retrieval*, pages 188–198. Springer, 2015.

Vladimír Boža, Broňa Brejová, and Tomáš Vinař. DeepNano: Deep Recurrent Neural Networks for Base Calling in MinION Nanopore Reads. Technical Report arXiv:1603.09195, arXiv preprints, 2016.

James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC Bioinformatics*, 13(1):238, 2012.

Rayan Chikhi, Guillaume Rizk, et al. Space-efficient and exact de bruijn graph representation based on a bloom filter. In *WABI*, pages 236–248, 2012.

Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de bruijn graphs. In *International Conference on Research in Computational Molecular Biology*, pages 35–55. Springer, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. Technical Report 1412.3555, arXiv, 2014.

David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1998.

Scott C. Clark, Rob Egan, Peter I. Frazier, and Zhong Wang. ALE: a generic assembly likelihood evaluation framework for assessing the accuracy of genome and metagenome assemblies. *Bioinformatics*, 29(4):435–443, 2013.

Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C. Boutros, and Jared T. Simpson. Nanocall: An Open Source Basecaller for Oxford Nanopore Sequencing Data. Technical Report doi:10.1101/046086, bioRxiv, 2016.

Nicolaas Govert de Bruijn and Paul Erdos. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

Arthur L Delcher, Adam Phillippy, Jane Carlton, and Steven L Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

Viraj Deshpande, Eric DK Fung, Son Pham, and Vineet Bafna. Cerulean: A hybrid assembly using high throughput short and long reads. In *Algorithms in Bioinformatics (WABI)*, volume 8126 of *LNCS*, pages 349–363. Springer, 2013.

RW Eglese. Simulated annealing: a tool for operational research. *European Journal of Operational Research*, 46(3):271–281, 1990.

Adam C English, Stephen Richards, et al. Mind the gap: upgrading genomes with Pacific Biosciences RS long-read sequencing technology. *PLoS One*, 7(11):e47768, 2012.

Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13: 12, 2009.

John Gallant, David Maier, and James Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.

Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Mohammadreza Ghodsi, Christopher M Hill, Irina Astrovskaya, Henry Lin, Dan D Sommer, Sergey Koren, and Mihai Pop. De novo likelihood-based measures for comparing genome assemblies. *BMC Research Notes*, 6(1):334, 2013.

Sante Gnerre, Iain MacCallum, et al. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Symposium on Experimental Algorithms (SEA)*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Szymon Grabowski and Marcin Raniszewski. Sampling the suffix array with minimizers. In *International Symposium on String Processing and Information Retrieval*, pages 287–298. Springer, 2015.

Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer-Verlag, Heidelberg, Germany, 2012.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Dark knowledge. *Presented as the keynote in BayLearn*, 2014.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. Technical Report 1503.02531, arXiv, 2015.

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. ART: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

Martin Hunt, Taisei Kikuchi, Mandy Sanders, Chris Newbold, Matthew Berriman, and Thomas D Otto. Reapr: a universal tool for genome assembly evaluation. *Genome Biology*, 14(5):R47, 2013.

Illumina. *E.coli* MG1655 Illumina sequencing dataset. ftp://webdata:webdata@ussd-ftp.illumina.com/Data/SequencingRuns/MG1655/MiSeq_Ecoli_MG1655_110721_PF.bam, 2015. Accessed: 2015-03-03.

Guy Joseph Jacobson. *Succinct static data structures.* PhD thesis, 1988.

G David Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

Kim Judge, Simon R Harris, Sandra Reuter, Julian Parkhill, and Sharon J Peacock. Early insights into the potential of the Oxford Nanopore MinION for the detection of antimicrobial resistance genes. *Journal of Antimicrobial Chemotherapy*, 70(10):2775–2778, 2015.

M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 98–107. Springer, 2003.

Haim Kaplan and Nira Shafrir. The greedy algorithm for shortest superstrings. *Information Processing Letters*, 93(1):13–17, 2005.

Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer, 2003.

John D Kececioglu and Eugene W Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13(1-2):7–51, 1995.

John Dimitri Kececioglu. *Exact and approximation algorithms for DNA sequence reconstruction.* PhD thesis, 1991.

David R Kelley, Michael C Schatz, Steven L Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):R116, 2010.

W James Kent. Blat—the blast-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature Biotechnology*, 30(7):693–700, 2012.

Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *bioRxiv*, page 071282, 2017.

Peter Kuljovský. Efficient substring search in long sequence. slovak: Efektívne vyhľadávanie krátkych reťazcov v dlhej sekvencii. Bachelor's thesis, Comenius University, 2016.

Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

C Lee Giles, Gary M Kuhn, and Ronald J Williams. Dynamic recurrent neural networks: Theory and applications. *IEEE Transactions on Neural Networks*, 5(2):153–156, 1994.

Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. Technical Report 1303.3997, arXiv, 2013.

Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

Yang Li, Pegah Kamousi, Fangqiu Han, Shengqi Yang, Xifeng Yan, and Subhash Suri. Memory efficient minimum substring partitioning. In *Proceedings of the VLDB Endowment*, volume 6, pages 169–180. VLDB Endowment, 2013.

Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. *BioMed Research International*, 2012, 2012.

Nicholas J Loman. `https://s3.climb.ac.uk/nanopore/E_coli_K12_1D_R9.2_SpotON_2.tgz`, 2016.

Nicholas J Loman, Joshua Quick, and Jared T Simpson. `http://www.ebi.ac.uk/ena/data/view/ERR1147230`, 2015a.

Nicholas J Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature Methods*, 12(8):733–735, 2015b.

Matthew Loose, Sunir Malla, and Michael Stout. Real-time selective sequencing using nanopore technology. *Nature Methods*, 13(9):751–754, 2016.

Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *Siam Journal on Computing*, 22(5):935–948, 1993.

Paul Medvedev, Son Pham, Mark Chaisson, Glenn Tesler, and Pavel Pevzner. Paired de bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. *Journal of Computational Biology*, 18(11):1625–1634, 2011.

Alexander S Mikheyev and Mandy MY Tin. A first look at the Oxford Nanopore MinION sequencer. *Molecular Ecology Resources*, 14(6):1097–1102, 2014.

Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

Satomi Mitsuhashi, Kirill Kryukov, So Nakagawa, Junko S Takeuchi, Yoshiki Shiraishi, Koichiro Asano, and Tadashi Imanishi. A portable system for metagenomic analyses using nanopore-based sequencer and laptop computers can realize rapid on-site determination of bacterial compositions. *bioRxiv*, page 101865, 2017.

Eugene W Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 1995.

Eugene W Myers. Efficient local alignment discovery amongst noisy long reads. In *Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

Niranjan Nagarajan, Timothy D Read, and Mihai Pop. Scaffolding and validation of bacterial genome assemblies using optical restriction maps. *Bioinformatics*, 24(10):1229–1235, 2008.

Alexis L Norris, Rachael E Workman, Yunfan Fan, James R Eshleman, and Winston Timp. Nanopore sequencing detects structural variants in cancer. *Cancer Biology & Therapy*, 17(3):246–253, 2016.

Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Workshop on Algorithms Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.

Jaroslav Petrucha. Efficient representation of set of short strings. slovak: Efektívna reprezentácia množiny krátkych reťazcov. Master's thesis, Bachelor's thesis, Comenius University, 2016.

Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

Nicolas Philippe, Mikael Salson, Thierry Lecroq, Martine Leonard, Therese Commes, and Eric Rivals. Querying large read collections in main memory: a versatile data structure. *BMC Bioinformatics*, 12(1):242, 2011.

Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu. A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. *BMC Genomics*, 13(1):341, 2012.

Joshua Quick, Philip Ashton, Szymon Calus, Carole Chatt, Savita Gossain, Jeremy Hawker, Satheesh Nair, Keith Neal, Kathy Nye, Tansy Peters, Elizabeth De Pinna, Esther Robinson, Keith Struthers, Mark Webber, Andrew Catto, Timothy J Dallman, Peter Hawkey, and Nicholas J Loman. Rapid draft sequencing and real-time nanopore sequencing in a hospital outbreak of Salmonella. *Genome Biology*, 16:114, 2015.

Joshua Quick, Nicholas J Loman, Sophie Duraffour, Jared T Simpson, Ettore Severi, Lauren Cowley, Joseph Akoi Bore, Raymond Koundouno, Gytis Dudas, Amy Mikhail, Nobila Ouedraogo, Babak Afrough, Amadou Bah, Jonathan H. J Baum, Beate Becker-Ziaja, Jan Peter Boettcher, Mar Cabeza-Cabrerizo, Alvaro Camino-Sanchez, Lisa L Carter, Juliane Doerrbecker, Theresa Enkirch, Isabel Garcia-Dorival, Nicole Hetzelt, Julia Hinzmann, Tobias Holm, Liana Eleni Kafetzopoulou, Michel Koropogui, Abigael Kosgey, Eeva Kuisma, Christopher H Logue, Antonio Mazzarelli, Sarah Meisel, Marc Mertens, Janine Michel, Didier Ngabo, Katja Nitzsche, Elisa Pallasch, Livia Victoria Patrono, Jasmine Portmann, Johanna Gabriella Repits, Natasha Y Rickett, Andreas Sachse, Katrin Singethan, Ines Vitoriano, Rahel L Yemanaberhan, Elsa G Zekeng, Trina Racine, Alexander Bello, Amadou Alpha Sall, Ousmane Faye, Oumar Faye, N'Faly Magassouba, Cecelia V Williams, Victoria Amburgey, Linda Winona, Emily Davis, Jon Gerlach, Frank Washington, Vanessa Monteil, Marine Jourdain, Marion Bererd, Alimou Camara, Hermann Somlare, Abdoulaye Camara, Marianne Gerard, Guillaume Bado, Bernard Baillet, Deborah Delaune, Koumpingnin Yacouba Nebie, Abdoulaye Diarra, Yacouba Savane, Raymond Bernard Pallawo, Giovanna Jaramillo Gutierrez, Natacha Milhano, Isabelle Roger, Christopher J Williams, Facinet Yattara, Kuiama Lewandowski, James Taylor, Phillip Rachwal, Daniel J Turner, Georgios Pollakis, Julian A Hiscox, David A. Matthews, Matthew K O'Shea, Andrew McD. Johnston, Duncan Wilson, Emma Hutley, Erasmus Smit, Antonino Di Caro, Roman Wolfel, Kilian Stoecker, Erna Fleischmann, Martin Gabriel, Simon A Weller, Lamine Koivogui, Boubacar Diallo, Sakoba Keita, Andrew Rambaut, Pierre Formenty, Stephan Gunther, and Miles W Carroll. Real-time, portable genome sequencing for Ebola surveillance. *Nature*, 530(7589):228–232, 2016.

Atif Rahman and Lior Pachter. CGAL: computing genome assembly likelihoods. *Genome Biology*, 14(1):R8, 2013.

Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 22(3):557–567, 2012.

David Sankoff and Joseph B Kruskal. Time warps, string edits, and macromolecules: the theory and practice of sequence comparison. *Reading: Addison-Wesley Publication, 1983, edited by Sankoff, David; Kruskal, Joseph B.*, 1, 1983.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45(11):2673–2681, 1997.

Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

Jared T Simpson, Rachael Workman, Philip C Zuzarte, Matei David, Lewis Jonathan Dursi, and Winston Timp. Detecting dna methylation using the oxford nanopore technologies minion sequencer. *bioRxiv*, page 047142, 2016.

Martin Smith. Nanoporetech community. `https://community.nanoporetech.com/posts/ultra-long-read-validation`, 2016.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

Z Sweedyk. A 2.5-approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 2000.

Niko Välimäki and Eric Rivals. Scalable and versatile k-mer indexing for high-throughput sequencing data. In *Bioinformatics Research and Applications (ISBRA)*, Lecture Notes in Bioinformatics, pages 237–248. Springer, 2013.

Zhong Wang, Mark Gerstein, and Michael Snyder. Rna-seq: a revolutionary tool for transcriptomics. *Nature reviews genetics*, 10(1):57–63, 2009.

Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, 2014.

WTC Human Genetics. MinION sequence data for clinical gram-negatives. `https://www.ebi.ac.uk/ena/data/view/SAMEA3713789`, 2016.

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.