UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2467

RAPID OVERLAPPING OF SINGLE MOLECULE HIGH-FIDELITY SEQUENCING DATA

Suzana Pratljačić

Zagreb, June 2021

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2467

RAPID OVERLAPPING OF SINGLE MOLECULE HIGH-FIDELITY SEQUENCING DATA

Suzana Pratljačić

Zagreb, June 2021

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 12 March 2021

MASTER THESIS ASSIGNMENT No. 2467

Student:	**Suzana Pratljačić (0036498021)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Mile Šikić

Title: **Rapid overlapping of Single Molecule High-Fidelity Sequencing Data**

Description:

Third-generation sequencing technologies facilitated the genome assembly problem significantly due to their ability to read considerably longer fragments than their predecessors. The only drawback is the high error rate present in such fragments, although algorithms were adapted quickly to tolerate it. Recently, Pacific Biosciences improved their sequencing chemistry and sequencer performance, which resulted in a novel protocol enabling high-fidelity (HiFi) fragments. The fragment length distribution of the protocol has a mean around 25 kbp without heavy tails, but the most remarkable thing is the accuracy, which is higher than 99%. The fact that the data has a small number of errors can be used to devise algorithms that will adapt less sensitive approaches. Higher accuracy enables decreasing the time needed to find overlaps between all pairs of fragments or between fragments and a reference genome. The main goal of this thesis is to adapt a publicly available overlapping algorithm for long erroneous reads or implement a novel one that will be more suited for this type of data. The solution should be appropriate for parallel architectures and implemented in C++. The source code has to be well documented using comments and should follow the Google C++ Style Guide when possible. The complete application should be hosted on GitHub under an OSI-approved license.

Submission date: 28 June 2021

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2467

Pristupnica:	**Suzana Pratljačić (0036498021)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Mile Šikić

Zadatak: **Brzo preklapanje visoko pouzdanih jednomolekularnih očitanja**

Opis zadatka:

Tehnologije treće generacije uređaja za sekvenciranje značajno olakšavaju problem sastavljanja genoma zbog mogućnosti očitanja znatno duljih fragmenata u odnosu na prethodnike. Jedini nedostatak je visoka razina pogreške prisutna u takvim fragmentima, iako su se algoritmi brzo prilagodili da ju toleriraju. Nedavno, tvrtka Pacific Biosciences je poboljšala korištene kemikalije i performanse uređaja za očitanje što je rezultiralo novim protokolom koji proizvodi visoko pouzdane fragmente. Srednja vrijednost distribucije duljine fragmenata ovoga protokola je oko 25 kbp bez teških repova, no najimpresivnija je točnost od iznad 99%. Činjenica da podaci imaju mali broj pogrešaka može se koristiti za osmišljavanje algoritama manje osjetljivih na pogrešku. Visoka točnost omogućuje smanjenje vremena potrebnog za pronalazak preklapanja između parova fragmenata ili fragmenata i referentnog genoma. Glavni cilj ove teze je prilagodba javno dostupnih algoritama za preklapanje dugačkih greškovitih očitanja ili osmišljavanje i implementacija novoga algoritma koji će biti bolje prilagođen ovom tipu podataka. Rješenje mora biti pogodno za paralelnu arhitekturu i implementirano u jeziku C++. Izvorni kod treba biti iscrpno dokumentiran koristeći komentare i slijediti Google C++ Style Guide kada je to moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednom od OSI odobrenih licenci.

Rok za predaju rada: 28. lipnja 2021.

I am grateful to my mentor Mile Šikić for his invaluable guidance and support throughout this project.

I want to thank my family and friends, especially my mother and father, whose love and support made me who I am.

CONTENTS

1. Introduction

Over the last fifty years, a lot of research has been done in developing technologies for DNA and RNA sequencing, resulting in three well-known sequencing generations. The focus of the first generation was the accuracy of the fragments, which resulted in accurate but short fragments. The second generation has increased data availability by introducing cheap and fast sequencing. Long error-prone fragments produced by third-generation sequencing technologies have significantly accelerated studies on genomes, but a high error rate still limits many downstream applications.

In 2019, Pacific Bioscience introduced highly accurate long-read sequencing (HiFi sequencing), a paradigm that combines the best concepts from traditional short and long error-prone reads technologies. The HiFi reads are characterized by lengths between 10 and 25 kb on average and greater than 99.5% base-level resolution. These accurate long reads provide valuable information that can be used to improve many genome studies. A prerequisite for all downstream analysis is to assemble fragments produced by sequencing technologies. Tools that can map reads to the existing reference genome (find the segment on the reference genome that is most similar to the fragment) are essential for that step.

Ever since Pacific Bioscience announced these new accurate reads, there has been a growing interest in finding and developing mapping algorithms adjusted to HiFi reads. The hypothesis of this article is that characteristics of HiFi data enable utilizing approaches that are less sensitive to errors and consequently reduce time consumption while increasing the accuracy of results.

Several mapping and aligning tools have already been developed and adjusted for HiFi data. Minimap2 (Li, 2018) has recently been extended with a preset for PacBio HiFi/CCS genomic fragments, enabling optimal performance and high accuracy. Another popular tool, developed on top of the Minimap2 codebase, is Winnowmap. Winnowmap (Jain et al., 2020) improves mapping accuracy by optimizing the algorithm to perform better on highly repetitive sequences.

The described tools are based on minimizers and use the standard seed-chain-align

procedure. Roberts et al. proposed the original idea behind minimizers in 2004, and ever since then, they have been used by most string comparison tools in bioinformatics. Simply put, minimizers are short substrings of a larger string. Algorithms for minimizer sampling often guarantee that a similar set of minimizers will be chosen on similar strings. Representing the reference genome with a set of minimizers reduces storage requirements while enabling fast search by storing these minimizers in hash tables. In the seed-chain-align procedure, seeds are minimizers extracted from fragments with the same minimizer sampling algorithm used in sampling minimizers from the reference genome. A key step in fragment mapping by this method is to find exact matches between the minimizers taken from the fragments and the minimizers from the reference genome. These exact matches are found efficiently by searching the beforementioned hash table. Exact matches represent *anchors*. Chaining multiple anchors approximates the mapping of the whole fragment. The chaining step is often conducted by dynamic programming inspired by the longest increasing subsequence algorithm.

The drawback of the described approaches are many false-positive anchors, causing the chaining time to be dominant in the execution (Babojelić, 2020). The false-positive anchors often occur due to tandem repeats and repetitive regions in the reference genome. Tandem repeats are short patterns of nucleotides that are repeated multiple times in the reference. The repetitive regions are long near-identical sequences of nucleotides that appear numerous times. Usually, to improve the time consumption, Minimap2 discards minimizers that occur too often, causing poor mapping quality within repetitions. The second improvement suggested in the paper Overlapping Single-Molecule High-Fidelity Sequencing Data (Babojelić, 2020) is to extract a smaller number of minimizers in order to improve the execution time, which results in losing sensitivity.

In this article, we will present the novel algorithm for mapping fragments against reference genome based on suffix arrays. Utilizing suffix arrays enables the development of different sampling algorithms that can capture critical information and preserve sensitivity while reducing the number of samples. This paper focuses on mapping fragments against the reference sequence, but we will propose methods to adjust the algorithm for finding overlaps between fragments. We will analyze the performance and accuracy of the proposed algorithm on both simulated and real datasets.

2. Data

2.1. Data formats

FASTA and FASTQ formats are *de-facto* standards for storing nucleotide and peptide sequences. The fundamental building units of biological sequences, amino acids, are represented as single-letter codes in these text-based formats.

There are two types of lines in FASTA format. The description line regularly contains the sequence name and often includes additional information like a sequence identifier. The lines of sequence data must follow the description line. Symbol '>' at the beginning of the description line allows distinguishing between line types. The lines in FASTA format are generally shorter than 80 characters, which is also recommended by the norm.

FASTQ is an extended FASTA format that contains the corresponding quality scores in addition to the sequence and its description (Cock et al., 2010). The qualities, as well as amino acids, are stored as single-character codes. There are four different types of lines in FASTQ format. The first line, which can be recognized by the '@' symbol at the beginning, contains the nonoptional sequence identifier and the optional description. The line with sequence data is the second line. The third line, beginning with the '+' character, may contain supplementary information like the sequence identifier and description. The fourth line stores quality scores for the sequence represented by the second line. The quality score can take on a value between the lowest and highest quality. According to Phred quality, the lowest quality is 33 ('!' in ASCII), and the highest quality is 126 ('$\sim$' in ASCII).

The quality score describes a probability that the corresponding amino acid is incorrectly sequenced. The standard equation which associates error probability and quality is $Q = -10 \log_{10} p$.

PAF format, an output of many mapping tools, is a simple text format describing the approximate mapping positions between sequences. Each line represents one alignment or overlap. The line in PAF format is TAB-delimited and contains the fields

described in Table 2.1.

Table (2.1) PAF format description, Taken from Minimap2 manual page.

Column	Type	Description
1	string	Query sequence name
2	int	Query sequence length
3	int	Query start (0-based; BED-like; closed)
4	int	Query end (0-based; BED-like; open)
5	char	Relative strand: "+" or "-"
6	string	Target sequence name
7	int	Target sequence length
8	int	Target start on original strand (0-based)
9	int	Target end on original strand (0-based)
10	int	Number of residue matches
11	int	Alignment block length
12	int	Mapping quality (0-255; 255 for missing)

2.2. PacBio HiFi Data

Single Molecule, Real-Time (SMRT) Sequencing is Pacific Bioscience technology that enables long-read sequencing. PacBio third-generation long reads are produced by one pass of the enzyme around the circular template. HiFi reads creation require multiple passes of enzyme around the circular template in order to achieve high accuracy. Enzyme passes generate many subreads, and the consensus over them is called a HiFi read. The following datasets, provided by Pacific Bioscience, are used in this paper.

Table (2.2) The PacBio HiFi reads

Dataset	Reference size	Subreads
Human chromosome 19	61707364	141750
Human chromosome 13	113566686	241583

2.3. Simulated Data

When developing new bioinformatics tools, it is essential to experiment with various methods for different algorithm components. The algorithm should be evaluated from several points to assess how the included method affects the solution of a particular problem. After evaluation, it is easy to decide whether to incorporate the considered methods in the final algorithm.

Additionally, when developing new algorithms, it is crucial to compare their performance with existing solutions. It is critical to consider the execution time and the accuracy of the obtained results when comparing performance between different tools. However, it is difficult to use real data in the evaluation process since the error and alignment information are unavailable. Therefore, it is required to simulate the data to know the ground truth and enable the evaluation process.

Since there is no available simulator adjusted to HiFi data, we have developed a simple simulator. The developed simulator can perform two different functions - sequence generation and fragment generation.

Optimizing mapping tools to perform well in practice is challenging because of the uneven distribution of minimizers in most genomes. Genomes contain repeats that cause some minimizers to appear too often and carry very little information for the mapping process. Minimap2 discards the most frequent minimizers to decrease time complexity. A consequence is that the accuracy in repetitive areas of genomes is reduced too. However, tools that map fragments against reference must work correctly regardless of the proportion of repetitions in the reference genome.

Sequence generation supported by the developed simulator helps optimize the algorithm to perform well in repetitive areas since it can create artificial repeats. Simple control of the proportion of repetitive regions and their positions allows simple monitoring of the tool execution in areas of interest. Different options offered by the sequence simulator are shown below.

```
usage: ./Refgen [options ...] <ref_size> [instructions ..]

  # default output is stdout
  <ref_size>
    the size of the reference
  instructions
   instructions for repetitive segments creation
   list of (index seed) values
```

```
options:
  -r, --repetitive-counter <int>
    default: 0
    the number of instructions
  -s, --seed <int>
    random seed for reference generation
  -p, --probability <double>
    the probability that the base will be mutated,
    to make repetitive regions distinguishable
  -n, --name <string>
    reference name
  -h, --help
    prints the usage
```

The following command will create the sequence with two equal segments, first runs from index 0 to index 10 exclusively, and second from index 20 to index 30.

```
./Refgen -r 4 -p 0 50 0 5 10 17 20 5 30 58
```

The result of executing the previous command is the following sequence.

```
>ARTIFICIAL_REF
AGTCGAACCGAATTAGAGGTAGTCGAACCGAGTGTGGCCAAAAAAGTCAT
```

Two simulators for PacBio data have inspired the implementation of the fragment simulator. PBSIM2 (Ono et al., 2020) simulate the non-uniformity of quality scores with a generative model for quality scores based on a hidden Markov model. PBSIM2 also implements another type of simulation called sampling-based simulation. The idea behind sampling simulation is to randomly choose fragments from the real dataset and use their characteristics to simulate new fragments. In the simulation, the new fragment is extracted from the reference, and errors are introduced according to the quality scores of the chosen fragment. Although interesting for its simplicity, the disadvantage of this approach is that it ignores the fact that there are low-quality regions on the reference. The second simulator that has influenced the development of our reads simulation procedure is PaSS (Zhang et al., 2019). The approach that PaSS implements is based on learning patterns from real sequencing data. First, to learn error distribution, PaSS aligns provided reads to the reference genome. PaSS

learns the probability for different error types considering the context. The context is considered by counting for all possible substrings (substrings are usually small in size) the number of occurrences of different errors types inside the given substring.

Our reads simulator uniformly extracts nucleotide sequences from a given reference. The sequencing errors (substitutions, insertions, and deletions) are simulated according to one of two implemented error models. The first, straightforward model defines the sequencing error as uniform distribution, meaning that the error probability is equal for every position of every generated read.

The second model involves preprocessing to learn quality distribution. The quality is learned for each nucleotide throughout the reference. Preprocessing requires subreads and their approximate alignment to the given reference. Needleman-Wunsch algorithm is employed for each subread to calculate the number of matched nucleotides between the reference genome and subread. If the ratio of matched nucleotides and the alignment length is lesser than the predefined threshold, the subread is discarded. Otherwise, the optimal alignment is used to update the quality distribution. For each position in reference, the quality is calculated as the average of qualities obtained as follows:

1. Insertion has occurred if the nucleotide from the subread has no counterpart in the reference. The quality of the previous position in nucleotide is updated with the lowest possible quality.

2. The deletion has occurred if the nucleotide from reference has no counterpart in the subread. The quality of the current position in reference is updated with the lowest possible quality.

3. If the nucleotides from reference and subread are equal for the given position, the quality is updated with the corresponding quality in the subread.

4. If the nucleotides are not matched, the quality is updated with the lowest possible quality.

If the position in reference is not covered with any subread and consequently has no quality information, the quality for that position is the highest possible.

The probability of the error is calculated from the quality, based on the Phred quality definition.

$$p = 10^{\frac{-Q}{10}} \tag{2.1}$$

Each error type occurs with equal probability. Deletion and substitution work as expected. In the case of insertion, half of the inserted nucleotides are randomly

chosen. The first next nucleotide in the reference determines the other half of inserted nucleotides, as explained in the PBSIM2 paper (Ono et al., 2020). The options for simulating reads are shown below.

```
usage: ./Readsgen [options ...] <reference_path>
        [<reads_path>, <paf_path>]

  # default output is stdout
  <reference_path>
    path to the reference
  <subreads_path>
    path to the subreads of the reference,
    required when model is set
  <alignment_path>
    path to the alignements paf file

  options:
    -m, --model
      error model will be trained from reads and alignements
     -l, --read_length <int>
      default: 25000
      the length of the reads
    -n, --num_reads <int>
      default: 10000
      the number of fragments to be generated
    -c, --complement
      default: 0.5
      proportion of reverse complement
    -p, --error_probability <double>
      default: 0.01
      probability of base sequencing error
    -s, --seed <int>
      random seed
    -h, --help
      prints the usage
```

The following command will generate 5 subreads of size 10 from the previously

simulated reference. The probability of the sequencing error is 0.1. The probability that the simulated subread is reverse complement equals 0.5.

```
./Readsgen -l 10 -n 5 -c 0.5 -p 0.1 reference.fasta
```

The result of the previous command is shown below.

```
@ARTIFICIAL_REF_0_TARGET_SIZE_10_STRAIN_+_POSITION_000000001
GGTCCGAAGG
+
~!~~!~~~!!
@ARTIFICIAL_REF_1_TARGET_SIZE_10_STRAIN_-_POSITION_000000035
TTTTTTGGCC
+
~~~~~~~~~~
@ARTIFICIAL_REF_2_TARGET_SIZE_10_STRAIN_-_POSITION_000000032
TTTGGCCACA
+
~~~~~~~~~~
@ARTIFICIAL_REF_3_TARGET_SIZE_10_STRAIN_+_POSITION_000000037
CCAAAAAAGT
+
~~~~~~~~~~
@ARTIFICIAL_REF_4_TARGET_SIZE_10_STRAIN_+_POSITION_000000024
CAACCGAGTG
+
!~~~~~~~~~
```

3. Methods

Mapping assembly is a standard technique for reconstructing the original sequence from fragments produced by sequencing technologies when the reference genome exists. Tools that map fragments to the existing reference are crucial for this assembly type. Since sequencing technologies do not produce completely accurate fragments, these tools must perform a non-exact matching algorithm to map fragments to the reference.

A possible definition of the approximate string matching problem is to find one substring of the reference that has the smallest *edit distance* to the given fragment among all substrings of the reference. The edit distance is defined as the smallest number of primitive operations required to match the fragment and reference exactly. Primitive operations are usually deletion, substitution, and insertion.

An efficient solution to the approximate string matching problem uses dynamic programming (DP). The Smith-Waterman algorithm is a DP approach that aims to divide the matching problem into the smallest subproblems and use their solutions to find the optimal local alignment between fragment and reference. The idea is to calculate the minimum edit distance between the first i characters in the fragment and any substring that ends at position j in the reference for each pair of ending indices. When calculating the minimum value for the pair (i, j), the previous values are used. The time complexity of this approach is $\mathcal{O}(np)$, where n is the reference size, and p is the fragment size.

The $\mathcal{O}(np)$ time complexity makes the before-mentioned approaches useless in bioinformatics. In recent years, the most popular algorithms for approximate string matching are those based on reducing the number of possible approximate matches by extracting substrings[1] from the fragment and matching these substrings to the reference with an exact matching algorithm. Although the most common choice for an algorithm that performs exact matching between the fragment and reference substrings is a hash-based algorithm, the method we will describe uses a suffix array.

[1] In bioinformatics, these substrings are usually called k-mers.

3.1. Overview of k-mer based methods

The proposed method is inspired by several rapid approximate sequence comparison methods that have been developed over the past few years. These methods heavily rely on k-mers and k-mer hash functions to provide state-of-the-art performance. Following, we will give an overview of these methods and describe how they influenced the development of the proposed algorithm.

One of the first such methods was BLAST (Altschul et al., 1990). BLAST utilizes the k-mer hash function to find potential matches, which are later expanded by dynamic programming to produce the final mapping. The algorithm first iterates through the reference and stores hash values for k-mers at positions $1, w+1, 2w+1...$ in the hash table. Potential matches are found by calculating the hash values for all k-mers in the fragment and searching the hash table for exact matches.

Minimap2 (Li, 2018) uses minimizers instead of sequentially sampled k-mers. Minimizers are defined as the smallest k-mers in a window containing a predefined number of consecutive k-mers of the sequence. A formal definition of minimizers and the sampling algorithm can be found in the paper "Reducing storage requirements for biological sequence comparison" (Roberts et al., 2004). Minimizers are collected in linear time regarding the sequence length and indexed using the hash table. The key in the hash table is the minimizer hash value. The corresponding value is a set of positions in the sequence. Given the fragment sequence and minimizers indexed using the hash table, Minimap2 finds exact matches between minimizers sampled from the fragment and minimizers stored in the hash table by comparing their hash values. These exact matches are called anchors according to the Minimap2 paper, and we will use that notion throughout the paper. An anchor is defined as 3-tuple (x, y, w), indicating that the k-mer of length w at position x in the reference is equal to k-mer of the same size w at position y in the fragment. The obtained anchors are chained using the chaining procedure, inspired by the longest increasing subsequence.

The described tools have considerably influenced the development of HiFiMapper, the tool for mapping fragments against reference based on a suffix array. Like these k-mer based methods, our tool samples substrings from the given fragment in order to find anchors. We have implemented several sampling algorithms, and one of them is equal to the procedure BLAST uses to extract k-mers from reference. Anchors obtained by finding exact matches using the suffix array are chained with the procedure introduced by Minimap2. The main difference is that HiFiMapper creates a suffix array to enable fast matching instead of storing the reduced k-mer representation of

the reference sequence.

3.2. Overview of the proposed method

In this section, we will give an overview the basic steps of the proposed method and the motivation behind them. We will briefly describe each component and later, through the chapter, provide the reader with detailed explanations and implementation choices that have been made.

The essential part of the proposed method is the suffix array, a space-efficient data structure that enables quickly locating all occurrences of the pattern[2] within the reference. A suffix array stores all suffixes of the reference sequence in sorted order. Suffixes are usually stored as starting positions of suffixes in the reference sequence, making memory requirements insignificant.

Finding an occurrence of the pattern within the reference corresponds to finding a suffix whose prefix is equal to that pattern. Since suffix arrays stores suffixes in lexicographical order, all suffixes beginning with the given pattern can be easily found with two binary searches.

Given two sets of sequences, target[3] and queries[4], the proposed algorithm first builds suffix array for the target sequence. Although this algorithm is primarily designed for mapping fragments to a reference, it can be easily extended for mapping to multiple references (or finding overlaps between multiple fragments). Multiple targets can be joined with a special character that is not included in any sequence, after which a suffix array can be constructed for the resulting string.

After preprocessing is done and fast pattern matching is enabled, the algorithm samples the given queries. Using the suffix array as a method for exact matching allows us the implementation of various sampling strategies. Since the suffix array stores the complete reference, a sample could be any substring from the query. The first and simplest sampling method is to choose substrings randomly. The second sampling algorithm guarantees high query coverage by sampling consecutive substrings like BLAST. The third method, called informed sampling, extracts the samples that are likely to determine the correct query position in the reference. An additional structure, the longest common prefix, must be constructed to allow the third sampling method.

In addition to allowing flexibility in query sampling, the use of the suffix array

[2]The pattern is a string whose occurrence is searced in the suffix array.

[3]A target is a reference to which fragments are mapped.

[4]Queries are fragments that are mapped.

also enables much larger samples than the hash approaches. Approaches based on hash functions require relatively small samples because the larger the minimizer, the more likely it is that the sequencing error is involved. Also, an increase in the size of minimizers needs to be accompanied by an increase in the hash table size to remedy the collision probability. Suppose the minimizer contains at least one wrong nucleotide base. In that case, the minimizer cannot be correctly matched using hash values because two substrings must be completely equal to have equal hash values (if we have a perfect hash function). On the other hand, using suffix arrays allows matching only a prefix of the sample, meaning that we can still avoid a false negative if the sequencing error occurs inside the sample. Using large samples can significantly reduce search space, and therefore runtime. If the patterns are long enough, they can be spread over small repetitive parts such as tandem repeats.

Matching only a prefix of the pattern provides resistance to sequencing errors. However, using suffix arrays also provides another way to avoid sequencing errors. When searching the suffix array, it is possible to ignore nucleotide bases that are considered to be of poor quality. However, it is impossible to avoid deletion and insertion by using quality scores.

Although the described sampling strategies and the searching of the suffix array reduce the number of false-positive and false-negative matches, the obtained matches can still be scattered due to the presence of larger repetitions. So, in the end, it is necessary to chain the anchors in order to find the best alignment.

To summarize, we will outline the described basic steps:

1. Construct a suffix array (and longest common prefix array if necessary) for the given target.
2. Sample each query with one of the following algorithms:
   (a) Random sampling.
   (b) A sampling of consecutive substrings.
   (c) Informed sampling.
3. Search the suffix array to find anchors.
4. Chain the obtained anchors to find the best alignments.

In the following sections, we will explain each of the previous steps in more detail.

3.3. Suffix array

Suffix arrays are one of the most used data structures in string processing. They have achieved popularity in practice because of their simplicity and space compactivity when compared to suffix trees.

For some string S of length n, the $i-th$ suffix S_i of S is the substring $S[i, ..., n-1]$. A suffix array is an array of indices corresponding to lexicographically sorted starting positions of all suffixes.

As an example, we will look at the string S = *AACGAACGT$*. The string usually ends with a unique character $, called sentinel. The sentinel is defined to be the lexicographically smallest character.

All suffixes of the given string are listed below.

0. AACGAACGT$
1. ACGAACGT$
2. CGAACGT$
3. GAACGT$
4. AACGT$
5. ACGT$
6. CGT$
7. GT$
8. T$
9. $

When lexicographically sorted, the listed suffixes are in the following order:

9: $
0: AACGAACGT$
4: AACGT$
1: ACGAACGT$
5: ACGT$
2: CGAACGT$
6: CGT$
3: GAACGT$
7: GT$
8: T$

And finally, the suffix array of the given string will be (9,0,4,1,5,2,6,3,7,8).

There exist a plethora of suffix array construction algorithms. These algorithms differ significantly in time and space complexity, essential properties considering the increasing number of large-scale applications. The time complexity of the implemented algorithm called Induced Sorting Variable-Length LMS-Substrings (SA-IS) is linear in the reference size. The main idea of the SA-IS algorithm is to use the recursively obtained solution to the reduced problem to solve the original problem. In this paper, we will not present the details of the algorithm for the suffix array construction since the pseudocode of the algorithm, along with proves of its correctness, can be found in the paper "Two Efficient Algorithms for Linear Time Suffix Array Construction" (Nong et al., 2011).

3.4. Suffix array search

Once the suffix array is constructed, an $\mathcal{O}(p \cdot logn)$ pattern matching algorithm can be easily implemented (p is the length of the pattern, and n represents the size of the suffix array). The idea of the pattern matching algorithm is based on the fact that if the pattern is a substring of the string, then the pattern is the prefix of at least one suffix of that string. Since the suffix array contains all sorted suffixes, performing two binary searches can find all pattern occurrences. Although more efficient ways of searching suffix arrays can be implemented as described in the article "Replacing suffix trees with enhanced suffix arrays" (Abouelhoda et al., 2004), we have decided to use binary search to experiment easily at this stage of the research.

Binary search is an algorithm that can find the value in a sorted array in logarithmic time. In each step, binary search compares the target value with the middle element in the range. If the value is not equal to the middle element, the range is reduced by discarding the part where the value cannot be found. Therefore, half of the range is discarded in each step. The interval reduction continues until the value is found or the interval becomes empty.

Binary search is applied letter by letter. Each time a letter is matched, the range of possible suffixes could be reduced, and the search continues until all letters in the pattern are used. If the letter cannot be found in the current range, it means that the given pattern does not appear in the reference. If the suffix array is constructed for genomic data, this may be due to sequencing errors. However, if a significant portion of the prefix is found within the suffix array, the range from the step before trying to match sequencing error can be retained. In this way, it is possible to use large samples, and at the same time, avoid many false-negative matches.

Utilizing binary search to find patterns occurrences within the suffix array allows using quality information provided in FASTQ files. Ignoring nucleotide bases with low-quality represents a great advantage of the suffix array approach over the approaches that work with the hash functions. Nucleotide bases that are likely to be erroneously sequenced can be skipped, and the sample can be matched although it contains errors. For each nucleotide base in FASTQ format, a quality score can be used to calculate error probability. If the quality threshold is set beforehand, nucleotide bases can be labeled as poor-quality if the corresponding quality score is lesser than the quality threshold or as good-quality otherwise. Quality information can be used in a binary search by ignoring nucleotides identified as poor quality. For example, if the pattern $ACCT$ is to be matched and there exists information that an error occurred on the third letter, then the binary search looks for the pattern $AC * T$ (* represents any character). The described approach is implemented by not reducing the range of suffixes when the binary search hits a nucleotide base with poor quality. The wrong nucleotide base is skipped, and the search continues with the following nucleotide.

Ignoring nucleotide bases with poor quality does not help if a deletion or insertion occurs. However, in that case, it is still possible to match only the prefix - the part of the sample before the deletion. Also, homopolymer compression can be helpful to avoid insertions since, a nucleotide equal to the previous one is often inserted.

Following the example of Minimap2, our implementation allows setting the algorithm parameters so that samples that appear too frequently in the reference are discarded. If the range of suffixes contains more suffixes than the predefined threshold, none of these matches will be declared as an anchor.

Although it contributes significantly to performance, discarding matches often causes poor mapping accuracy for fragments sequenced from the repetitive regions. To improve mapping accuracy for genomes with repetitions, HiFiMapper implements a new heuristic called extended search. The heuristic allows reducing the number of anchors while preserving valuable information. If the algorithm finds more matches than the predefined threshold allows, the algorithm keeps extending the pattern and searching suffix array until the interval is reduced enough. In that way, the algorithm tries to find a difference that may be crucial in discovering the correct position.

3.5. Sampling

To detect potential matches between target and queries, the proposed method first finds candidate matches by extracting and matching substrings from queries. Perhaps the

easiest way to sample substrings is to extract every consecutive substring in the query. If w denotes the sample length and n represents the query size, then we would take the following substrings $S[0, ..., w-1]$, $S[1, .., w]$, $...S[n-w, .., n-1]$. Even though the described sampling approach allows high resistance to errors, it is not useful in practice due to the high runtime caused by many samples.

The other possible approach is to extract samples randomly. Although there is no guarantee that such sampling guarantees finding candidate matches that will result in optimal alignment, it shows promising results in practice.

Inspired by the BLAST strategy for taking k-mers from the reference, we have implemented another sampling procedure. If w denotes the sample length and n represents the query size, the beforementioned procedure will extract following substrings $S[0, ..., w-1], S[w, .., 2w-1], ...S[n-w, .., n-1]$.

The idea behind the third sampling method, called informed sampling, is to extract a small number of samples that carry the most information for determining the proper position of the query. The low query coverage reduces the time required to search samples in the suffix array, consequently reducing the total execution time.

The additional data structure, the longest common prefix array (LCP), is used to implement this sampling method. In the next section, we will explain the longest common prefix array and how it is used to implement an informed sampling algorithm.

3.5.1. Longest Common Prefix Array

Udi Manber and Gene Myers introduced the longest common prefix array (LCP array) to speed up the pattern matching algorithm. The LCP array is a data structure that augments the suffix array. The LCP array stores the values of the longest common prefixes between consecutive suffixes stored in the suffix array. If S represents string for which the suffix array SA is constructed, $LCP[i]$ represents the value of the longest common prefix between suffixes $S_{SA[i]}$ and $S_{SA[i+1]}$.

For example, the LCP array for the previously constructed suffix array:

9: $
0: AACGAACGT$
4: AACGT$
1: ACGAACGT$
5: ACGT$
2: CGAACGT$
6: CGT$
3: GAACGT$
7: GT$
8: T$

is (0,4,1,3,0,2,0,1,0).

The input for the LCP construction algorithm is the suffix array and the sequence for which it is built. Kasai's algorithm for LCP construction is used in our implementation.

The construction algorithm is based on a simple fact. If it is known that the longest common prefix between neighboring suffixes in the suffix array, denoted as S_i and S_j ($S_i \leq$S$_j$), is k letters long, $k > 0$, it can be concluded that the LCP between suffixes S_{i+1} and S_{j+1} is $k-1$ letters long. That is valid because suffixes S_{i+1} and S_{j+1} are obtained by removing the first letter from the suffixes S_i and S_j. Kasai's algorithm iterates through suffixes from longest to shortest to utilize the beforementioned fact and reuse the already calculated value k. However, S_{i+1} and S_{j+1} may not be adjacent in the suffix array, so it is not always possible to use the calculated value directly. The suffix S_{i+1} must be lexicographically smaller or equal than the suffix S_{j+1} (because S_i and S_j, have the same first letter and S_i is immediately before S_j in sorted suffix array), and there could be an arbitrary number of other suffixes between them. The longest common prefix between two suffixes not adjacent in the suffix array corresponds to the minimum of all values stored in the LCP array between these two suffixes. Therefore, all LCP values between the suffixes S_{i+1} and S_{j+1} in the LCP array are at least $k-1$.

Although LCP stores the longest common prefixes only for those suffixes that are adjacent in the suffix array, it can be easily used to find the longest common prefix for any two suffixes. The longest common prefix of any two suffixes is denoted by $lcp(S_i, S_j)$.

Suppose that S_i and S_j are not adjacent in the suffix array and that S_i is lexicographically smaller or equal to the S_j. The longest common prefix between these suffixes S_i and S_j is $min(LCP[IS[S_i]], LCP[IS[S_i]+1], ...LCP[IS[S_j]-1])$ where IS is inverse suffix array (IS contains for each suffix S_k its position in the suffix

array).

This means that for some suffix S_i the value of $lcp(S_i, S_j)$ is less than or equal to $lcp(S_i, S_k)$ if $IS[S_i] < IS[S_k] < IS[S_j]$. If $IS[S_i] > IS[S_k] > IS[S_j]$ then $lcp(S_i, S_k)$ is greater than or equal to $lcp(S_i, S_j)$. Simply put, it is to be expected that some suffix S_i has larger longest common prefixes with suffixes that are closer to that suffix in the suffix array. For example, suppose we wanted to find 10 suffixes that have the largest longest common prefixes with some suffix S_i among all suffixes. In that case, we could consider only 10 suffixes that are in front of the suffix S_i in the suffix array and 10 suffixes that are behind the suffix S_i in the suffix array.

Suppose that $lcp(S_i, S_j)$ is k. In that case, the first letter that distinguishes the suffix S_i from the suffix S_j is $S[i + k + 1]$.

For each suffix S_i, it is possible to find at which positions are the letters that distinguish this suffix S_i from its mos *similar* suffixes (suffixes that have the largest longest common prefix with the given suffix among all suffixes in the string). Suppose we want to find the letters that distinguish each suffix from its most similar N suffixes. Then we need to consider only N suffixes that precede the S_i in the suffix array and N suffixes after the S_i in the suffix array. Among these $2 \cdot N$ suffixes, the most similar N suffixes to suffix S_i are those that have the largest longest common prefixes with the given suffix S_i. The values of these longest common prefixes determine the letters that distinguish the suffix S_i from its most similar N suffixes.

Suppose that some suffix S_i is located in a repetitive region that appears at x more places in the sequence. Also, suppose that these repetitive regions are not identical but that every pair of these repetitive regions differs in one letter, which is not located at the beginning of the repetitive region. In this case, we expect that it is very likely that the most similar x suffixes to the suffix S_i are the suffixes located at the positions that represent the beginnings of these repetitive regions. The longest common prefixes between S_i and its most similar x suffixes reveal the position of the letters that can distinguish these repetitive regions. On the other hand, if the suffix S_i is not part of a repetitive region, we expect that the longest common prefixes between S_i and its most similar suffixes are relatively small numbers.

Information about letters that can distinguish repetitive regions cannot be used directly for sampling. Instead, it is first necessary to detect potential fragment positions in the reference with a small number of random samples. If the fragment is not part of a repetitive region, then that small set of random samples is sufficient to determine the correct position.

On the other hand, if the fragment is part of a repetitive region, then the random

samples determine the candidate positions. For each candidate position, it is then possible to find the distance to the letters that distinguish the suffix that starts at that position from its most similar suffixes. If we were to extract a sample that should be mapped to that distinguishing letter, its mapping could eliminate a large proportion of the candidate positions that are not part of the correct mapping. The position of that sample can be found based on the position of the sample that is mapped to the candidate position and the distance to the letter that distinguishes repetitive regions. The informed sampling algorithm samples precisely those samples.

To summarize, informed sampling first performs preprocessing to find letters that distinguish each suffix in the reference from similar suffixes. In the implementation, the positions are taken so that samples that should be mapped do not overlap. After the preprocessing is done, the candidate positions are determined using a small number of random samples. If the fragment is not part of a repetitive region, that set of random samples will determine the mapping. Also, there will probably be no distinguishing letters for these candidate positions since the longest common prefixes between suffixes that begin on candidate positions and those most similar to them are often smaller than the sample size (because the fragment is not part of the repetitive region). On the other hand, if the fragment is part of a repetitive region, the algorithm extracts new samples.

3.6. Chaining

The chaining algorithm implemented in this paper is described in the Minimap2 article. Although Minimap2 uses minimizers of equal sizes, all equations consider minimizer length and can be used when the anchors are not of equal lengths. All equations presented in this section are taken from the Minimap2 paper (Li, 2018). The algorithm input is a list of anchors sorted according to the end positions in the reference. The algorithm finds various possible chains of anchors and their chaining scores.

The maximal chaining score $f(i)$ up to anchor i of size w_i can be calculated with dynamic programming according to the equation:

$$f(i) = \max\{\max_{i>j\geq 1}\{\ f(j) + \alpha(j,i) - \beta(j,i)\}, w_i\} \tag{3.1}$$

The number of matching nucleotide bases between two anchors is represented by $\alpha(i,j) = \min\{\min\{y_i - y_j, x_i - x_j\}, w_i\}$. The gap cost is represented as $\beta(j,i)$. The

gap cost is defined with the following equation:

$$\beta(j,i) = \begin{cases} \infty & y_j \geq y_i \\ \infty & \max\{y_i - y_j, x_i - x_j\} > G \\ \gamma_c((y_i - y_j) - (x_i - x_j)) & \text{otherwise} \end{cases} \tag{3.2}$$

The gap cost is infinity if the distance between the anchors is greater than the predefined parameter G. $\gamma_c(l)$ is the function determines the cost of the gap of length l.

$$\gamma(l) = \begin{cases} 0.01 * \overline{w} * |l| + 0.5 * \log_2 |l| & l \neq 0 \\ 0 & l = 0 \end{cases} \tag{3.3}$$

The $\overline{w}$ in the definition of γ_c represents the average value of anchors lengths.

Calculating chaining scores with dynamic programming according to the described equation has $O(N^2)$ time complexity, where N is the number of anchors. Minimap2 proposed a heuristic that improves the quadratic complexity of the algorithm.

The heuristic idea is not to consider all possible predecessors, but only h of them when calculating chaining scores, resulting in $O(hN)$ time complexity. This approach is reasonable since chaining to the predecessors of the anchor that is already chained often results in a lower score. The authors of Minimap2 suggest that the constant h should be set to 50.

Each anchor continues some chain or starts a new one. If the anchor starts a new chain, then the anchor is its own predecessor. If an anchor continues a chain, then its predecessor is the anchor at the end of that chain. Storing predecessors when calculating chain scores allows backtracking and chain identification.

Among all the possible chains obtained with backtracking, Minimap2 identifies primary chains. Primary chains are chosen not to overlap more than 50% on the query sequence.

In the case of several chains of comparable quality, we have decided to allow users to define which chains should be reported. However, chains that overlap significantly on the reference will be discarded.

4. Implementation

The implemented tool allows setting some parameters in order to achieve the best performance for a particular use case. Each parameter is briefly explained below, and their default values are shown.

```
usage: ./HiFimapper [options ...] <target> [<sequences>]
  # default output is stdout
  <target>
    path to the targets in FASTA/FASTQ format
  <sequences>
    path to the queries in FASTA/FASTQ format

  options:
    -t, --threads <int>l
      defaul: 8
      number of threads
    -l, --sample_length <int>
      default: 50
      the length of the samples
    -c, --sample_count <int>
      default: 20
      the number of samples extracted from each query
    -m, --min_match <double>
      default: 0.8
      percentage of the sample that must be mapped
      for the match to be valid
    -q, --quality <int>
      default: 90
      phred quality
    -f, --frequency <int>
```

```
  default: 10
  maximum number of matches
-b, --bandwidth <int>
  default: 10
  size of bandwidth in which sample hits can be chained
-g, --gap <int>
  default: 10000
  maximal gap between sample hits in a chain
-d, --discard <bool>
  default: false
  discarding matches that occur more times
  than the default frequency
-e, --extended_search <bool>
  default: false
  allows the extended search heuristics
-a, --sequential <bool>
  default: false
  blast like sampling algorithm
-i, --lcp_information <bool>
  deafult: false
  use lcp information for sampling
-N, --secondary_alignements <int>
  default: 5
  number of secondary alignements
-p, --ratio <double>
  deafult: 0.8
  secondary to primary alignements ratio
-n, --minimal_anchors <int>
  default: 3
  minimal number of anchors on chain
-x, --lcp_search_size <int>
  defult: 100
  number of suffixes / 2 to calculate lcp
-h, --help
  prints the usage
```

The `threads` parameter allows defining the number of threads that parallelly execute suffix array search and chaining procedures. The `sample_length` parameter defines the size of the samples extracted from the query, and the `sample_count` parameter determines the number of these samples. The `min_match` parameter specifies the percentage of the sample that must be exactly matched to retain matches. Setting the `min_match` parameter to a value less than 1 allows using larger samples. The `quality` parameter sets a quality threshold. If the quality score of the nucleotide base is lesser than that threshold, then the base is marked as a poor quality base.

The `frequency` parameter determines the maximum size of a range of suffixes. If the `discard` parameter is set, then all samples that have more matches with a reference than the specified frequency will be discarded. Discarding anchors improves execution time, but at the same time, decreases mapping quality. If the `extended search` parameter is set, then the algorithm tries to reduce the number of matches by continuing the search outside the sample until the desired number of matches is reached or when it is no longer possible to expand and match the sample.

A random sampling algorithm is a default sampling method. If the `sequential` option is set, then a sequential sampling algorithm is used. Setting the `lcp_information` parameter allows using an informed sampling algorithm. In that case, an LCP array should be constructed for the given sequence. Then, for each position i in the reference sequence, the distances to letters that distinguish the suffix S_i from the most similar `lcp_search_size`$\cdot 2$ suffixes are found.

4.1. Dependencies

4.1.1. OpenMP

The OpenMP API (`https://www.openmp.org/`) supports multithreading in the C++ programing language. In C++, OpenMP uses #pragmas to fork additional threads and create constructs for work sharing. Work sharing construct allows splitting loop iterations among threads. OpenMP also supports synchronization mechanisms.

In the HiFiMapper implementation, OpenMP is used to parallelize several components of the algorithm. Suffix array can be searched in parallel without any synchronization mechanisms since binary search does not change underlying data. Also, each binary search is entirely independent of other searches. The work-sharing concept is utilized to distribute queries among multiple threads that implement the

same logic for finding anchors. In that way, each thread is in charge of finding anchors for several queries, which contributes significantly to performance.

The second component whose parallelization is straightforward is the chaining of the anchors. Anchors belonging to different queries are independent and consequently could be chained in parallel. The work-sharing concept is again an obvious choice to achieve the desired behavior.

4.1.2. Biosoup

Biosoup (`https://github.com/rvaser/biosoup`) is a C++ collection of header-only data structures implemented by Robert Vaser and used for storing bioinformatic sequences in various tools. In the HiFiMapper implementation, the NucleicAcid class was used and changed to support the required interface. NucleicAcid is implemented to improve memory usage by storing bases in two bits instead of 8-bit characters. The basic implementation saves the average quality for a block of 8 bases. We have changed the quality storing logic to support separate quality storage for each base and, at the same time, reduce memory usage. For each base, quality information is stored in one bit. The quality is set to 1 if the corresponding quality score is greater than the predefined quality threshold. In this way, the bases are divided into two groups, high-quality and low-quality. This information is later used so that bases belonging to the low-quality group would not be taken into account during the process of finding anchors.

4.1.3. Bioparser

Bioparser (`https://github.com/rvaser/bioparser`) is a C++ header-only parsing library developed by Robert Vaser. Except that it supports basic formats like FASTA and FASTQ, it also supports zlib compressed files. Parsing in batches enables easy memory management.

4.1.4. Python requirements

Python requirements are listed in the `requirements.txt` in the root directory of the project. It should be noted that Python is not required to run the mapper. Python and the listed requirements are used for writing experiments, as explained in Chapter 5. Requirements in this version are the following python packages: tabulate, matplotlib and pyfastx. The tabulate library allows printing tables in several formats, including

latex tables. The matplotlib package is used to draw sequence coverage graphs. The pyfastx package is used to simply read files with fragments to find those that are not mapped.

5. Results

In this chapter, we will evaluate the HiFiMapper and compare the performance with various existing tools when possible.

5.1. Evaluation

We have implemented a simple benchmark framework to facilitate the evaluation process and allow easy reproduction of all conducted experiments. All experiments listed in this chapter can be found in the tests/benchmarks folder in the GitHub repository.

5.1.1. Benchmark Framework

Benchmark framework is written in Python programming language. It provides abstractions for invoking various mapping tools, including HiFiMapper, Winnowmap and Minimap2. The framework comes with a simple API for specifying various options offered by the listed tools. Additionally, it is possible to generate and simulate references and reads using the Reference and Reads classes that provide abstractions for the HiFi simulator described in the Data chapter. Once the tools are called, it is easy to access the obtained results and the information generated during the mapping process, like time and memory consumption. Benchmark framework also offers evaluators for calculating mapping accuracy on results gained by mapping fragments generated by the HiFi simulator.

The mapping of the fragment is said to be correct if the Jaccard similarity between the calculated interval (defined by start and end positions in a reference written in the PAF output of tool) and true interval (true start and end positions of the fragment in the reference) is greater than or equal to 0.1. Jaccard similarity index is a measure of similarity defined as the ratio of the intersection and the union of two intervals. It describes the percentage of shared bases between the calculated and accurate interval.

The mapping accuracy can be calculated only for simulated fragments since there is no ground truth for real fragments. Therefore, in the experiments including real data, the evaluation metric is the number of mapped reads.

For each experiment, the command with which it is run is displayed. The command includes only parameters that differ from the default parameters given in Chapter 4.

5.1.2. Hardware

All experiments were performed on the same hardware, with the following specifications:

OS:	Ubuntu 20.04.2 LTS
Arhitecture:	x86_64
Processor:	AMD EPYC 7662 64-Core Processor
Cores	128
Memory:	738 GiB

5.2. Artificial Data

We have evaluated the influence of different parameters on the performance of HiFiMapper. Experiments in this section were performed on artificially generated datasets to conclude how parameters affect mapping accuracy and execution time.

Each table for each pair of parameters contains three performance indicators: mapping accuracy, number of unmapped fragments, and fragment mapping time. Mapping accuracy is calculated as the proportion of accurately mapped fragments. The fragment mapping time includes only the searching time (pattern matching in suffix array) and chaining time (chaining of the found matches). The fragment mapping time does not include preprocessing time, like suffix array construction and loading data, since it does not depend on the parameters whose influence has been studied in this section.

The first experiment shows how the sample length and the number of samples affect mapping quality and execution time. In this experiment 1×10^5 fragments of size 25×10^3 were mapped against a reference of size 25×10^6. There are no repetitive regions in the reference. It could be expected that the increase in mapping time follows the increase in sample size since pattern matching has $\mathcal{O}(plogn)$ time complexity, where p is the size of the pattern, and n is the size of the suffix array. Also, more

samples should take more time to find and chain matches. The data in the Table 5.1 confirm these run-time assumptions.

Table (5.1) The influence of `sample_length` (l) and `sample_count` (c) parameters on performance. The first number represents mapping accuracy (proportion of correctly mapped fragments). The second number represents the number of unmapped fragments. The third number is total fragment mapping time in seconds (without preprocessing). The experiment is started with the `HiFiMapper -t 256 -d -n 1 -N 1` command, and the values of l and c are set according to the table.

l / c	10	20	50	70	100
25	0.999, 0, 0.98	1.0, 0, 1.3	1.0, 0, 2.29	1.0, 0, 3.01	1.0, 0, 4.02
50	0.999, 0, 1.03	1.0, 0, 1.29	1.0, 0, 2.48	1.0, 0, 3.3	1.0, 0, 4.5
75	0.999, 0, 1.05	1.0, 0, 1.44	1.0, 0, 2.74	1.0, 0, 3.59	1.0, 0, 5.0
100	0.999, 0, 1.02	1.0, 0, 1.51	1.0, 0, 2.96	1.0, 0, 3.86	1.0, 0, 5.32
150	0.998, 0, 1.1	1.0, 0, 1.67	1.0, 0, 3.28	1.0, 0, 4.52	1.0, 0, 5.98

The second experiment investigates how the presence of repetitive regions affects performance. For that purpose, the reference of size 25×10^6, in which 30×10^3 long repetitive segments cover 10% of the reference size, was generated along with 1×10^5 fragments sampled from reference with 0.1% sequencing error probability. Fragments are 25×10^3 bases long. Repetitive segments were generated so that all bases are the same within all repetitive segments, except for approximately every 10000th base. Although repetitive segments make the mapping task more difficult, the differences (every 10,000th nucleotide base) should lead the algorithm to correctly map fragments that are completely sampled from the repetitive segment.

The results of the experiment are shown in the Table 5.2. We can observe that when sample length and sample count are small numbers, incorrect mappings occur because the algorithm cannot distinguish between repetitive segments. With an insufficient amount of information due to low coverage and the discarding of the samples, some fragments may not be mapped.

Table (5.2) The influence of `sample_length` (l) and `sample_count` (c) parameters on performance when mapping against highly repetitive reference. The first number represents mapping accuracy (proportion of correctly mapped fragments). The second number represents the number of unmapped fragments. The third number is total fragment mapping time in seconds (without preprocessing). The experiment is started with the `HiFiMapper -t 256 -m 0 -f 3 -d -n 1 -N 1` command, and the values of l and c are set according to the table.

l / c	10	20	50	70	100
25	0.981, 1976, 0.92	0.991, 513, 1.33	0.999, 8, 2.45	1.0, 0, 3.25	1.0, 0, 4.36
50	0.988, 849, 0.94	0.998, 70, 1.38	1.0, 0, 2.66	1.0, 0, 3.55	1.0, 0, 4.77
75	0.992, 306, 0.97	0.999, 2, 1.44	1.0, 0, 2.95	1.0, 0, 3.79	1.0, 0, 5.23
100	0.995, 88, 1.03	1.0, 0, 1.56	1.0, 0, 3.08	1.0, 0, 4.15	1.0, 0, 5.64
150	0.998, 12, 1.15	1.0, 0, 1.75	1.0, 0, 3.63	1.0, 0, 4.72	1.0, 0, 6.71

The following experiment examines how extended search heuristics affect fragment mapping accuracy. The same data set as in the previous experiment, highly repetitive reference, is used. We expect that the extended search heuristics can find the key differences that determine the correct position. From the data shown in Table 5.3, we can conclude that the extended search heuristic positively contributes to mapping accuracy without significantly affecting the mapping time.

Table (5.3) The influence of `sample_length` (l) and `sample_count` (c) parameters on performance when mapping against highly repetitive reference and using extended search heuristic. The first number represents mapping accuracy (proportion of correctly mapped fragments). The second number represents the number of unmapped fragments. The third number is total fragment mapping time in seconds (without preprocessing). The experiment is started with the `HiFiMapper -t 256 -m 0 -f 3 -d -e -n 1 -N 1` command, and the values of l and c are set according to the table.

l / c	10	20	50	70	100
25	0.999, 0, 0.88	1.0, 0, 1.34	1.0, 0, 2.35	1.0, 0, 3.28	1.0, 0, 4.37
50	0.999, 0, 0.97	1.0, 0, 1.36	1.0, 0, 2.58	1.0, 0, 3.23	1.0, 0, 4.72
75	0.999, 0, 1.15	1.0, 0, 1.46	1.0, 0, 2.69	1.0, 0, 3.61	1.0, 0, 4.92
100	0.999, 0, 1.07	1.0, 0, 1.55	1.0, 0, 2.83	1.0, 0, 3.68	1.0, 0, 5.2
150	1.0, 0, 1.08	1.0, 0, 1.6	1.0, 0, 3.17	1.0, 0, 4.37	1.0, 0, 5.78

5.3. Simulated Data

In this section, we will evaluate HiFiMapper on datasets including real references and fragments that are simulated from these references using the described HiFi simulator. Homo sapiens chromosomes 13 and 19 are real references used in this section. A total of 600000 fragments of size 25000 were generated for each chromosome. The probability that a sequencing error occurred is the same for all bases in the fragment, and its value is 0.1

For each chromosome, the HiFiMapper was run with different parameters. Minimap2 and Winnowmap were run with default parameters for HiFi data. To make it easier to evaluate the tools, we consider only one mapping for one fragment for each tool. The mapping that is considered is the one with the highest quality, or the first printed in the case of multiple mappings with equal quality. The number of correctly mapped, incorrectly mapped, and unmapped fragments are shown for each tool. Additionally, the total execution time is measured for each experiment.

Table 5.4 shows how sample size and quality information affect the performance of HiFiMapper when mapping fragments on chromosome 19. The smaller number of wrong mappings when using larger samples confirms the assumption that larger samples better determine the accurate position in the reference. Also, using quality information improves mapping accuracy.

Results in Table 5.5 show that the use of extended search heuristics along with discarding matches and sequential sampling gives the best results for chromosome 19. Similar parameters, with a higher frequency, provide the best results for chromosome 13, as shown in Table 5.6.

Coverage graphs visually show fragments mapping. For each position in the reference, the value on the coverage graph corresponds to the number of fragments that cover that position. Ideally, each position should be covered with an equal number of fragments. For the simulated datasets, coverage graphs are shown separately for correctly and incorrectly mapped fragments in Figure 5.1 and in Figure 5.2.

Suppose we define that a correct mapping is the mapping that maps a fragment to the part of the reference from which the fragment is sequenced. In that case, we can observe that all the tools incorrectly map fragments to approximately equal parts of the reference. However, the beforementioned definition of correct mapping is challenging to apply to real genomic data. Real sequences often contain repetitive regions that are completely identical and larger than the fragment being mapped. So, if the sequenced fragment appears several times in the sequence due to identical repetitive regions, then

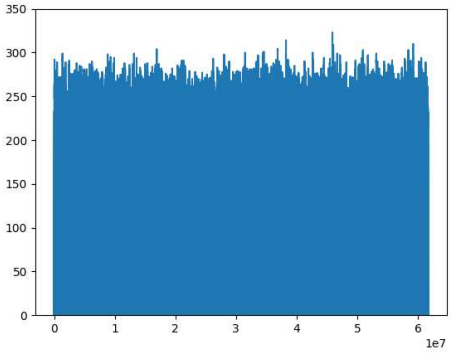
there are several equally good mappings. On the other hand, the algorithm is expected to find a correct mapping (mapping to the part from which the fragment was extracted), which is not possible in the described case. For example, human chromosome 13 has multiple completely identical repetitive segments larger than 25,000 bases. Since these segments are indistinguishable, all three tools *incorrectly* map fragments in these areas.

Table (5.4) Influence of `sample_size` and `quality` on HiFiMapper performance on dataset simulated from human chromosome 19. The reference size is 61707364. The total number of reads is 600000.

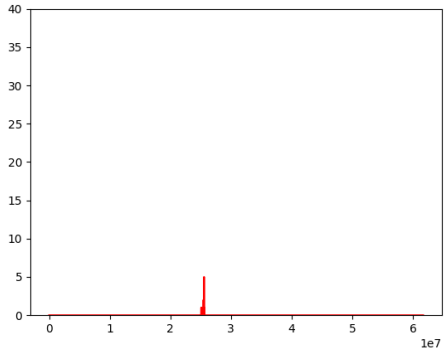
Method	RealTime	Correct	Wrong	Unmapped
HiFiMapper -t 256 -l 50 -c 1000 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	504.207288	595979	4021	0
HiFiMapper -t 256 -l 100 -c 500 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	385.435190	598009	1991	0
HiFiMapper -t 256 -l 250 -c 200 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	341.273078	599111	889	0
HiFiMapper -t 256 -l 500 -c 100 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	323.511647	599594	406	0
HiFiMapper -t 256 -l 50 -c 1000 -m 0 -f 15 -d -e -n 1 -N 1	465.228541	596229	3771	0
HiFiMapper -t 256 -l 100 -c 500 -m 0 -f 15 -d -e -n 1 -N 1	387.210408	598015	1985	0
HiFiMapper -t 256 -l 250 -c 200 -m 0 -f 15 -d -e -n 1 -N 1	346.227833	599096	904	0
HiFiMapper -t 256 -l 500 -c 100 -m 0 -f 15 -d -e -n 1 -N 1	327.387002	599617	383	0

Table (5.5) Influence of different parameters on HiFiMapper performance and comparison with other tools on dataset simulated from human chromosome 19. The reference size is 61707364. The total number of reads is 600000. The best results are highlighted for each tool. Only one mapping for each fragment was considered (the mapping with the highest mapping quality) in the evaluation process for each tool. If multiple mappings of one fragment have the same quality, the first printed mapping is selected.

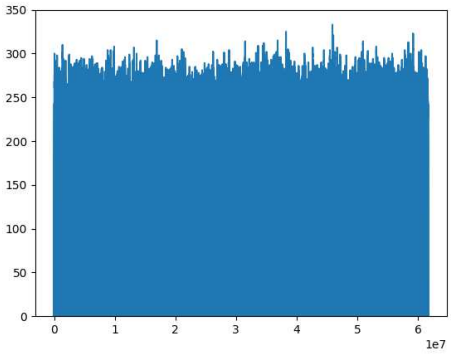
Method	RealTime	Correct	Wrong	Unmapped
HiFiMapper -t 256 -l 500 -f 15 -d -i -n 1 -N 1 -x 1000	319.091273	598879	1116	50
HiFiMapper -t 256 -l 500 -f 15 -d -i -n 1 -N 1 -x 5000	325.688057	598898	1099	48
HiFiMapper -t 256 -l 500 -f 100 -d -i -n 1 -N 1 -x 1000	322.700111	596248	3752	0
HiFiMapper -t 256 -l 500 -f 100 -d -i -n 1 -N 1 -x 5000	322.897989	596242	3758	0
HiFiMapper -t 256 -l 500 -c 50 -f 15 -d -i -n 1 -N 1 -x 1000	333.790990	598460	1540	3
HiFiMapper -t 256 -l 500 -c 50 -f 15 -d -i -n 1 -N 1 -x 5000	333.361971	598492	1508	1
HiFiMapper -t 256 -l 500 -c 50 -f 100 -d -i -n 1 -N 1 -x 1000	344.375467	594401	5599	0
HiFiMapper -t 256 -l 500 -c 50 -f 100 -d -i -n 1 -N 1 -x 5000	348.768581	594460	5540	0
HiFiMapper -t 256 -l 1000 -f 15 -d -i -n 1 -N 1 -x 1000	321.243638	597211	2789	515
HiFiMapper -t 256 -l 1000 -f 15 -d -i -n 1 -N 1 -x 5000	319.582608	597217	2783	468
HiFiMapper -t 256 -l 1000 -f 100 -d -i -n 1 -N 1 -x 1000	323.653853	596973	3027	479
HiFiMapper -t 256 -l 1000 -f 100 -d -i -n 1 -N 1 -x 5000	319.925540	597024	2976	460
HiFiMapper -t 256 -l 1000 -c 50 -f 15 -d -i -n 1 -N 1 -x 1000	341.150441	599219	781	1
HiFiMapper -t 256 -l 1000 -c 50 -f 15 -d -i -n 1 -N 1 -x 5000	337.311784	599231	769	3
HiFiMapper -t 256 -l 1000 -c 50 -f 100 -d -i -n 1 -N 1 -x 1000	338.858804	598770	1230	0
HiFiMapper -t 256 -l 1000 -c 50 -f 100 -d -i -n 1 -N 1 -x 5000	343.044332	598767	1233	1
HiFiMapper -t 256 -l 1000 -f 15 -d -e -a -n 1 -N 1	304.716986	599963	37	1
HiFiMapper -t 256 -l 1000 -f 15 -e -a -n 1 -N 1	305.441583	599963	37	1
HiFiMapper -t 256 -l 1000 -f 15 -d -a -n 1 -N 1	304.257653	599957	43	4
HiFiMapper -t 256 -l 1000 -f 15 -a -n 1 -N 1	300.641277	599959	41	1
HiFiMapper -t 256 -l 500 -f 15 -d -e -a -n 1 -N 1	311.444440	599990	10	0
HiFiMapper -t 256 -l 500 -f 15 -e -a -n 1 -N 1	306.527749	599982	18	0
HiFiMapper -t 256 -l 500 -f 15 -d -a -n 1 -N 1	308.270169	599983	17	0
HiFiMapper -t 256 -l 500 -f 15 -a -n 1 -N 1	312.909779	599941	59	0
HiFiMapper -t 256 -l 500 -f 15 -d -e -n 1 -N 1	299.494604	599721	279	3
HiFiMapper -t 256 -l 500 -f 15 -e -n 1 -N 1	298.865729	599774	226	0
HiFiMapper -t 256 -l 500 -f 15 -d -n 1 -N 1	300.738720	599309	680	137
HiFiMapper -t 256 -l 500 -f 15 -n 1 -N 1	302.649075	599556	444	0
minimap2 -x map-hifi -t 256 -N 0 -p 1	117.801	599994	6	0
Winnowmap -x map-pb -t 256 -p 1	517.977	599962	38	0



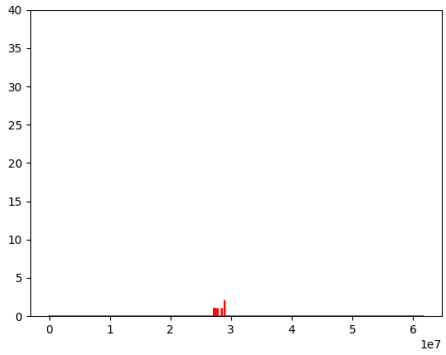
(a) Distribution of fragments correctly mapped against human chromosome 19 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 500 -f 15 -d -e -a -n 1 -N 1`.



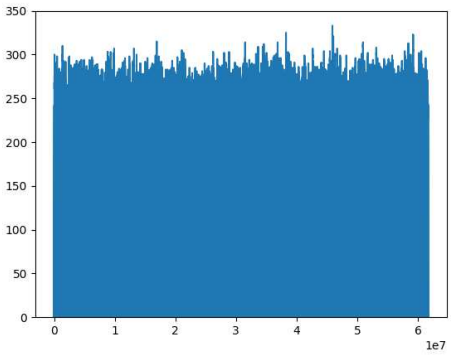
(b) Distribution fragments incorrectly mapped against human chromosome 19 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 500 -f 15 -d -e -a -n 1 -N 1`.



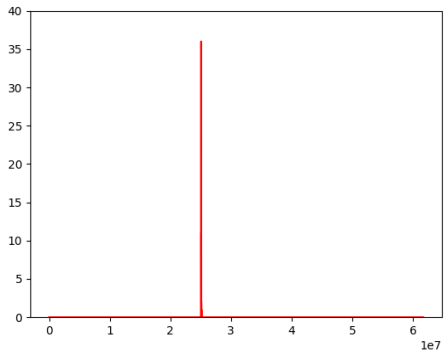
(c) Distribution of fragments correctly mapped against human chromosome 19 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 0 -p 1`.



(d) Distribution fragments incorrectly mapped against human chromosome 19 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 0 -p 1`.



(e) Distribution of fragments correctly mapped against human chromosome 19 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1`.

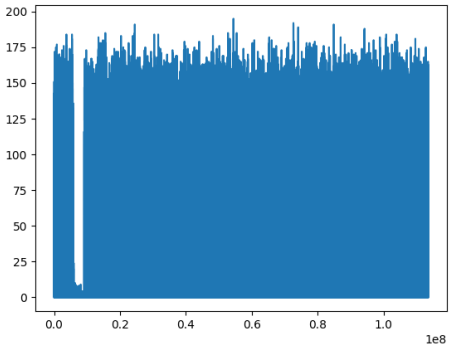


(f) Distribution fragments incorrectly mapped against human chromosome 19 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1`.

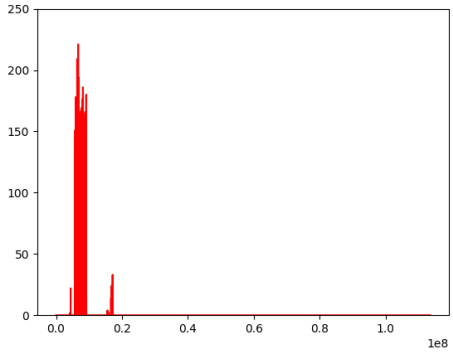
Figure (5.1) Coverage graphs for human chromosome 19

Table (5.6) Influence of different parameters on HiFiMapper performance and comparison with other tools on dataset simulated from human chromosome 13. The reference size is 113566686. The total number of reads is 600000. The best results are highlighted for each tool. Only one mapping for each fragment was considered (the mapping with the highest mapping quality) in the evaluation process for each tool. If multiple mappings of one fragment have the same quality, the first printed mapping is selected.

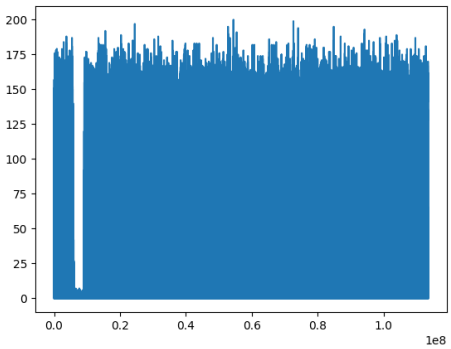
Method	RealTime	Correct	Wrong	Unmapped
HiFiMapper -t 256 -l 500 -f 75 -d -i -n 1 -N 1 -x 5000	413.182009	580316	19684	483
HiFiMapper -t 256 -l 500 -f 150 -d -i -n 1 -N 1 -x 5000	504.287402	580308	19692	108
HiFiMapper -t 256 -l 500 -c 50 -f 75 -d -i -n 1 -N 1 -x 5000	647.565967	580598	19402	44
HiFiMapper -t 256 -l 500 -c 50 -f 150 -d -i -n 1 -N 1 -x 5000	1156.483292	579598	20402	3
HiFiMapper -t 256 -l 1000 -f 75 -d -i -n 1 -N 1 -x 5000	393.827826	579525	20475	559
HiFiMapper -t 256 -l 1000 -f 150 -d -i -n 1 -N 1 -x 5000	400.320342	579068	20932	532
HiFiMapper -t 256 -l 1000 -c 50 -f 75 -d -i -n 1 -N 1 -x 5000	691.141032	581631	18369	9
HiFiMapper -t 256 -l 1000 -c 50 -f 150 -d -i -n 1 -N 1 -x 5000	791.671239	580956	19044	10
HiFiMapper -t 256 -l 1000 -f 75 -d -e -a -n 1 -N 1	328.569060	582378	17622	2
HiFiMapper -t 256 -l 1000 -f 75 -e -a -n 1 -N 1	327.946092	582416	17584	2
HiFiMapper -t 256 -l 1000 -f 75 -d -a -n 1 -N 1	333.072614	582236	17764	33
HiFiMapper -t 256 -l 1000 -f 75 -a -n 1 -N 1	328.382296	582373	17627	2
HiFiMapper -t 256 -l 1000 -f 75 -d -e -n 1 -N 1	333.697609	580416	19584	282
HiFiMapper -t 256 -l 1000 -f 75 -e -n 1 -N 1	335.362858	580614	19386	223
HiFiMapper -t 256 -l 1000 -f 75 -d -n 1 -N 1	326.541286	580069	19931	445
HiFiMapper -t 256 -l 1000 -f 75 -n 1 -N 1	334.440019	580535	19465	202
HiFiMapper -t 256 -l 500 -f 75 -d -e -a -n 1 -N 1	336.171272	582567	17433	0
HiFiMapper -t 256 -l 500 -f 75 -e -a -n 1 -N 1	334.481023	582562	17438	0
HiFiMapper -t 256 -l 500 -f 75 -d -a -n 1 -N 1	339.645863	582426	17574	17
HiFiMapper -t 256 -l 500 -f 75 -a -n 1 -N 1	341.340677	581189	18811	0
HiFiMapper -t 256 -l 500 -f 75 -d -e -n 1 -N 1	336.260026	581983	18017	12
HiFiMapper -t 256 -l 500 -f 75 -e -n 1 -N 1	310.353638	581928	18072	0
HiFiMapper -t 256 -l 500 -f 75 -d -n 1 -N 1	333.297837	579900	20100	1008
HiFiMapper -t 256 -l 500 -f 75 -n 1 -N 1	322.623996	579760	20240	0
minimap2 -x map-hifi -t 256 -N 0 -p 1	166.855	582418	17582	0
Winnowmap -x map-pb -t 256 -p 1	2793.503	582536	17464	0



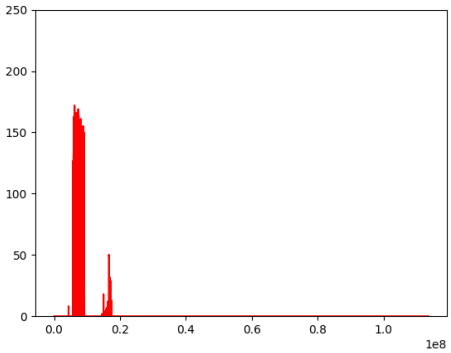
(a) Distribution of fragments correctly mapped against human chromosome 13 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 500 -f 75 -d -e -a -n 1 -N 1`.



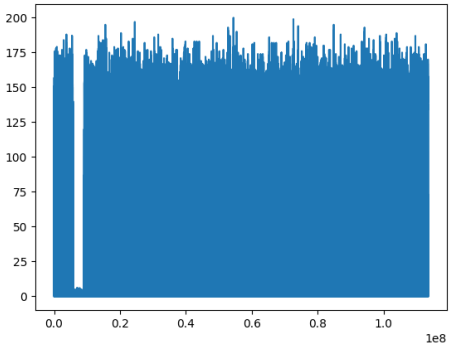
(b) Distribution fragments incorrectly mapped against human chromosome 13 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 500 -f 75 -d -e -a -n 1 -N 1`.



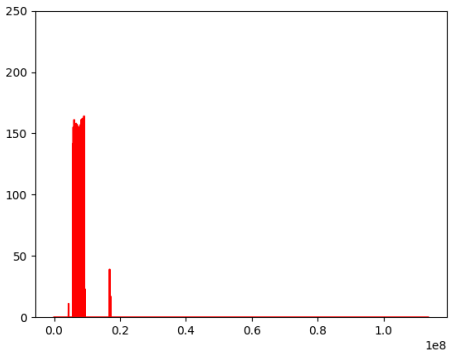
(c) Distribution of fragments correctly mapped against human chromosome 13 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 0 -p 1`.



(d) Distribution fragments incorrectly mapped against human chromosome 13 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 0 -p 1`.



(e) Distribution of fragments correctly mapped against human chromosome 13 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1`.



(f) Distribution fragments incorrectly mapped against human chromosome 13 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1`.

Figure (5.2) Coverage graphs for human chromosome 13

5.4. Real Data

In this section, we will compare the performance of HiFiMapper with the currently most used mapping tools on real data. Since real data do not include information about positions in reference from which fragments were sequenced, it is impossible to estimate the mapping accuracy. Therefore, only the number of unmapped fragments and the execution time are shown.

The HiFiMapper was run with several different sample sizes. The number of samples was set to cover each base in the fragment with about 5 samples. Random sampling was used for both data sets. The results for human chromosome 19 are shown in Table 5.7. It is noticeable that the run time increases as the sample size decreases, perhaps because smaller samples appear more frequently in the reference. Table 5.8 contains results for human chromosome 13.

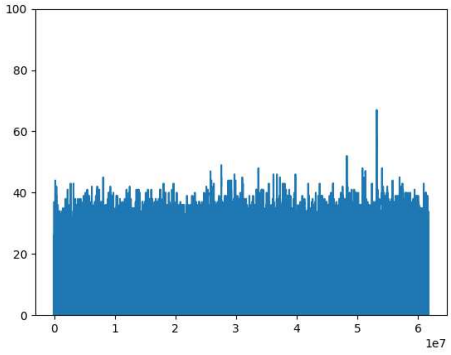
The coverage graphs in Figure 5.3 visually show the mapping results of all tools. Only one mapping, with the highest mapping quality, for each fragment was considered. If multiple mappings of one fragment have the same quality, the first printed mapping is selected.

Table (5.7) Influence of different parameters on HiFiMapper performance and comparison with other tools on human chromosome 19 and real fragments.

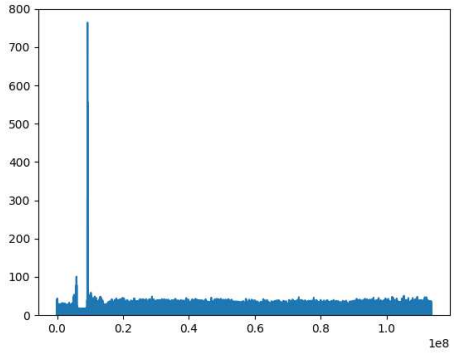
Method	RealTime	Unmapped
HiFiMapper -t 256 -l 50 -c 1000 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	127.440	0
HiFiMapper -t 256 -l 100 -c 500 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	86.992	0
HiFiMapper -t 256 -l 250 -c 200 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	75.803	0
HiFiMapper -t 256 -l 500 -c 100 -m 0 -q 0 -f 15 -d -e -n 1 -N 1	71.482	12
minimap2 -x map-hifi -t 256 -N 1 -p 1	28.822	18
Winnowmap -x map-pb -t 256 -p 1	66.847	30

Table (5.8) Influence of different parameters on HiFiMapper performance and comparison with other tools on human chromosome 13 and real fragments.

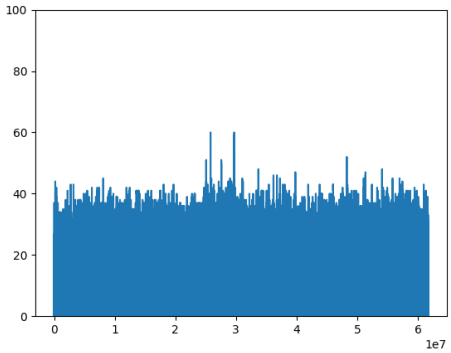
Method	RealTime	Unmapped
HiFiMapper -t 256 -l 50 -c 1000 -m 0 -q 0 -f 75 -d -e -n 1 -N 1	244.591	0
HiFiMapper -t 256 -l 100 -c 500 -m 0 -q 0 -f 75 -d -e -n 1 -N 1	168.191	0
HiFiMapper -t 256 -l 250 -c 200 -m 0 -q 0 -f 75 -d -e -n 1 -N 1	139.073	0
HiFiMapper -t 256 -l 500 -c 100 -m 0 -q 0 -f 75 -d -e -n 1 -N 1	139.213	5
minimap2 -x map-hifi -t 256 -N 1 -p 1	44.774	11
Winnowmap -x map-pb -t 256 -p 1	384.600	13



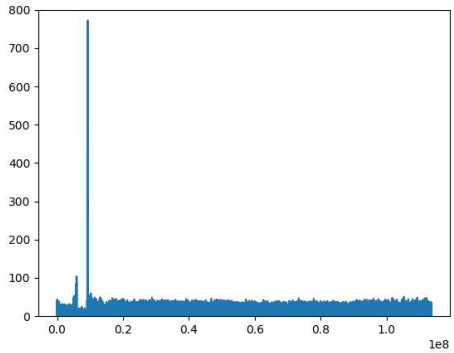
(a) Distribution of fragments mapped against human chromosome 19 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 250 -c 200 -m 0 -q 0 -f 15 -d -e -n 1 -N 1.`



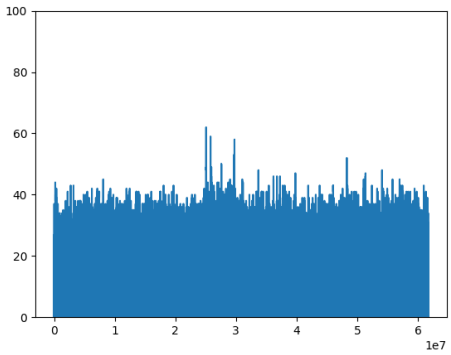
(b) Distribution of fragments mapped against human chromosome 13 with HiFiMapper. The data is generated with the following command `HiFiMapper -t 256 -l 250 -c 200 -m 0 -q 0 -f 75 -d -e -n 1 -N 1.`



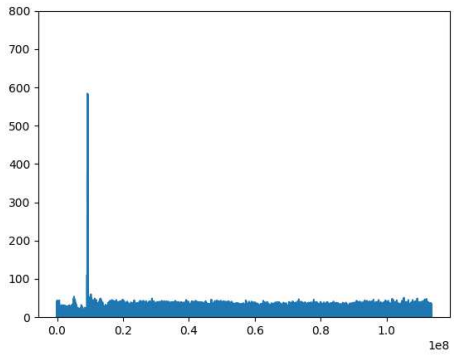
(c) Distribution of fragments mapped against human chromosome 19 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 1 -p 1.`



(d) Distribution of fragments mapped against human chromosome 13 with Minimap2. The data is generated with the following command `minimap2 -x map-hifi -t 256 -N 1 -p 1.`



(e) Distribution of fragments mapped against human chromosome 19 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1.`



(f) Distribution of fragments mapped against human chromosome 13 with Winnowmap. The data is generated with the following command `Winnowmap -x map-pb -t 256 -p 1.`

Figure (5.3) Coverage graphs for human chromosome 13 and human chromosome 19

6. Conclusion

This paper describes an algorithm for mapping fragments to a known reference based on a suffix array. The suffix array has many advantages over hash functions that are the most used alternative to the suffix array. The main advantage is that using a suffix array allows the implementation of various sampling methods. Another advantage is that the suffix array search procedure can use information about sequencing errors. Additionally, it is possible to match only the prefix of a sample, making it possible to find part of the sample even though there is an error in it. This high resistance to errors allows the use of much larger samples compared to the size of the minimizer, and larger samples often better determine the true position of the fragment in the reference.

We have provided numerous experiments on different datasets, including artificially generated data, simulated data, and real data. The HiFiMapper was run with various parameters to understand their impact on accuracy and execution time. The presented results show that the performance indicators are comparable to the currently most popular tools.

Since this algorithm is still a work in progress, it is not yet suitable for some typical bioinformatics applications. Future work includes optimizing implementation and exploring how the stated benefits of the suffix array can be utilized to reduce execution time and increase accuracy even more.

BIBLIOGRAPHY

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, i Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. ISSN 1570-8667. doi: https://doi.org/10.1016/S1570-8667(03)00065-0. URL https://www.sciencedirect.com/science/article/pii/S1570866703000650. The 9th International Symposium on String Processing and Information Retrieval.

Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, i David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. ISSN 0022-2836. doi: https://doi.org/10.1016/S0022-2836(05)80360-2. URL https://www.sciencedirect.com/science/article/pii/S0022283605803602.

D. Babojelić. *Overlapping Single Molecule High-Fidelity Sequencing Data*. Doktorska disertacija, 2020.

Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, i Peter M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, Apr 2010. ISSN 1362-4962. doi: 10.1093/nar/gkp1137. URL https://pubmed.ncbi.nlm.nih.gov/20015970. 20015970[pmid].

Chirag Jain, Arang Rhie, Nancy Hansen, Sergey Koren, i Adam M. Phillippy. A long read mapping method for highly repetitive reference sequences. *bioRxiv*, 2020. doi: 10.1101/2020.11.01.363887. URL https://www.biorxiv.org/content/early/2020/11/02/2020.11.01.363887.

Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34 (18):3094–3100, 05 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty191. URL https://doi.org/10.1093/bioinformatics/bty191.

Ge Nong, Sen Zhang, i Daricks Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *Computers, IEEE Transactions on*, 60:1471 – 1484, 11 2011. doi: 10.1109/TC.2010.188.

Yukiteru Ono, Kiyoshi Asai, i Michiaki Hamada. PBSIM2: a simulator for long-read sequencers with a novel generative model of quality scores. *Bioinformatics*, 37(5): 589–595, 09 2020. ISSN 1367-4803. doi: 10.1093/bioinformatics/btaa835. URL https://doi.org/10.1093/bioinformatics/btaa835.

M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, i J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, Srpanj 2004. doi: 10.1093/bioinformatics/bth408. URL https://doi.org/10.1093/bioinformatics/bth408.

Wenmin Zhang, Ben Jia, i Chaochun Wei. Pass: a sequencing simulator for pacbio sequencing. *BMC Bioinformatics*, 20(1):352, Jun 2019. ISSN 1471-2105. doi: 10.1186/s12859-019-2901-7. URL https://doi.org/10.1186/s12859-019-2901-7.

Brzo preklapanje visoko pouzdanih jednomolekularnih očitanja

Sažetak

Tvrtka Pacific Bioscience 2019. godine predstavila je novu tehnologiju sekvenciranja visoko pouzdanih fragmenata koja kombinira koncepte korištene za sekvenciranje tradicionalnih kratkih i dugačkih očitanja s greškama. Visoko pouzdani fragmenti karakterizirani su velikom duljinom i visokom točnosnu. Činjenica da podatci imaju jako mali broj grešaka može se koristiti za osmišljavanje algoritama koji su manje osjetljivi na pogreške. U ovom radu predstavit ćemo algoritam za mapiranje visoko pouzdanih fragmenata. Algoritam koristi sufiksno polje i seed-chain-align proceduru. Analizirat ćemo performanse i točnost razvijenog alata i usporediti ga sa suvremenim algoritmima za mapiranje.

Ključne riječi: Visoko pouzdana jednomolekularna ocitanja, sufiksno polje

Rapid overlapping of Single Molecule High-Fidelity Sequencing Data

Abstract

In 2019, Pacific Bioscience introduced highly accurate long-read sequencing (HiFi sequencing), a paradigm that combines concepts from traditional short and long error-prone reads technologies. Long lengths and high base-level resolution characterize the HiFi fragments. These characteristics can be utilized to design algorithms that are less sensitive to errors. In this article, we present the novel algorithm for HiFi reads mapping. The algorithm is based on suffix arrays and a standard seed-chain-align procedure. We will analyze the performance and accuracy of the developed tool and compare it to state-of-the-art algorithms.

Keywords: HiFi PacBio reads, DNA sequencing, suffix array