SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2467

BRZO PREKLAPANJE VISOKO POUZDANIH JEDNOMOLEKULARNIH OČITANJA

Suzana Pratljačić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2467

BRZO PREKLAPANJE VISOKO POUZDANIH JEDNOMOLEKULARNIH OČITANJA

Suzana Pratljačić

Zagreb, lipanj 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.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

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 first generation was focused on short and accurate reads. The second generation introduced cheap and fast sequencing. Long reads with a high error rate characterize the third generation.

Although third-generation sequencing have accelerated studies on the genome, the high error rate limits some 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 long lengths, 25kpb on average, and base-level resolution above 99.9% single-molecule read accuracy. These new read characteristics enable utilizing approaches that are less sensitive to errors, causing the reduction of computation time.

Several mapping and aligning tools have been developed and adjusted for HiFi reads. Minimap2 (Li (2018)), the universal sequence alignment tool, has recently been extended with a preset for PacBio HiFi/CCS genomic reads, enabling optimal performance and high accuracy for that use case. 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 utilized by most sequence alignment tools. They reduce the storage requirements and enable fast approximate matching. Minimizers are often indexed in a hash table where the key is a hash value of a minimizer. The position of a minimizer in the original sequence is the corresponding value. In the seed-chain-align procedure, the seeds are parts of the query sequence the algorithms are trying to match because they cannot match the whole query due to the sequencing errors and high time

complexity. When using minimizers, the seeds are usually minimizers collected in the query sequence. The exact matches between seeds and reference sequence are found by calculating seed hash and searching through the hash table. The values corresponding to the key determined by the seed hash are these exact matches, also called anchors, between parts of the query and parts of the reference. Since anchors represent places where some part of the reference is equal to the query, chaining multiple anchors could approximate the mapping of the whole query sequence. The chaining step is often conducted by dynamic programming inspired by the longest increasing subsequence algorithm.

The drawback of the described approach is many false-positive anchors, causing the chaining time to be dominant in the execution. [Babojelic] The false-positive anchors often occur due to tandem repeats and repetitive regions in the reference. Tandem repeats are short patterns of nucleotides that are repeated multiple times in the reference. The repetitive regions are longer near-identical sequences of nucleotides that appear numerous times in the reference. Usually, to improve the time complexity, the minimizers that are part of the repetitive region are discarded, causing poor mapping quality within these areas.

The drawback of the described approach is many false-positive anchors, causing the chaining time to be dominant in the execution. [Babojelic] The false-positive anchors often occur due to tandem repeats and repetitive regions in the reference. Tandem repeats are short patterns of nucleotides that are repeated multiple times in the reference. The repetitive regions are longer near-identical sequences of nucleotides that appear numerous times in the reference. Usually, to improve the time complexity, the minimizers that are part of the repetitive region are discarded, causing poor mapping quality within these areas.

The minimizers must be small in size, so they cannot resolve the tandem repeats. If we were to increase the size of the minimizer, then we would have a lot of false negatives because the anchors are found by comparing the hash values of the minimizers that do not tolerate sequencing errors. On the other hand, repetitive regions often differ in several nucleotide bases that are hard to capture with minimizers.

In this article, we will present the novel algorithm based on suffix arrays for sequence alignment and mapping. We will analyze the performance and accuracy compared to state-of-the-art tools.

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. 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 sequence alignment 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 2.2.

Tablica 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.

Tablica 2.2: The PacBio HiFi reads

Dataset	Reference	Subreads
Drosophila melanogaster	-	SRR12473480
Anopheles gambiae	GCF_000005575.2	SRX8642992 SRX8642991
Phlebotomus papatasi	-	SRR12454518
Oryza sativa	MH63RS2	SRP218375)

2.3. Simulated Data

When developing new bioinformatics tools, researchers experiment with various methods that should be evaluated from different points. After evaluation, it is easy to decide whether to include the considered methods in the final solution. Also, to compare the mapping accuracy between different tools, it is necessary to know the ground truth. It is difficult to use real data in the evaluation process since the error and alignment information are not easy to obtain. Since there is no available simulator adjusted to HiFi reads, we have developed a simple tool to simulate the data according to characteristics of real HiFi reads. The developed simulator can perform two different functions - sequence generation and read generation.

Sequence alignment tools must work correctly regardless of the proportion of repetitive regions. However, adjusting tools to perform well in repetitive areas is challenging. Sequence generation is helpful in that process since it allows the creation of artificial repeats. Simple control of the proportion of repetitive regions and their positions allows monitoring the tool execution in areas of interest.

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

  # default output is stdout
  <ref_size>
    the size of the reference you want to generate
  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
```

For example, 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
```

Reads generator uniformly extracts nucleotide sequence from a given reference. The sequencing errors, substitutions, insertions, and deletions, are simulated according to 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, during which quality is learned for each nucleotide throughout the reference. Preprocessing requires subreads and their approximate alignment to the given reference in the form of paf format. Needleman-Wunsch algorithm is employed for each subread to calculate the number of matched nucleotides between the reference 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. 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.

```
usage: ./Readsgen [options ...] <reference_path>
        [<subreads_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 readings to be generated
    -c, --complement
      default: true
      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 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. [PBSIM2]

3. Methods

The implementation of HiFiMapper 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.

One of the first such methods was BLAST Altschul et al. (1990). They utilize the k-mer hash function to find potential matches, which are later expanded by dynamic programming to produce the final mapping. They first iterate through the reference and store hash values for k-mers at positions 1, w+1, 2w+1 ... in the hash table. Potential matches are searched by calculating the hash values for all k-mers in the query and searching the hash table.

Minimap (Li (2018)) uses minimizers instead of k-mers and, like BLAST, stores their hash values in the hash table. Minimizers are defined as the smallest k-mers in a window containing consecutive k-mers of the sequence. Exact matches, determined by minimizers with equal hash values, between target and queries are called anchors. An anchor is a 3-tuple (x,y,w), indicating that the kmer of length w extracted at position x in the target is equal to kmer of the same size w at position y in the query. The obtained anchors are chained using an algorithm similar to that for solving the longest increasing sequence of the problem.

The developed tool is similar to the described approaches because it finds potential matches by extracting k-mers from the query and finding exact matches between k-mers and the target sequence. The difference between HiFiMapper and the k-mer based method is in finding these exact matches. HiFiMapper, instead of storing the reduced k-mer representation of the target sequence, creates the suffix array to enable fast finding of exact matches and at the same time allow high flexibility in selecting k-mers from the query.

The essential part of the developed sequence alignment tool is the suffix array, a space-efficient data structure used for fast searching patterns in the given string. Utilizing the suffix array enables us to create a different seeding strategy than the standard

one in the seed-chain-align approaches.

Since the suffix array stores the complete reference, seeds are not required to be extracted consistently, which sometimes could be time-consuming. In the presented algorithm, seeds are randomly extracted samples from the query sequence. The length of the sample is the algorithm parameter.

Matching a seed with the reference is performed by searching the suffix array. While searching the suffix array, we can stop the search before hitting the end of the pattern. That allows matching only a prefix of seed, meaning that if the sequencing error occurs in the seed, we can still avoid a false negative.

3.1. Suffix array construction

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 (reference) of length n, its suffix array is an array of indices corresponding to lexicographically sorted suffixes, denoted as SA(S).

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, described in the paper Two Efficient Algorithms for Linear Time Suffix Array Construction, is linear in the reference size.

The main idea of the Induced Sorting Variable-Length LMS-Substrings algorithm is to use the recursively obtained solution to the reduced problem to solve the original problem. For sake of simplicity, the given string should be terminated by the lexicographically smallest character, usually called sentinel. In the rest of the section, we will describe the basics of each step of the algorithm.

3.1.1. Reducing the Problme

The first step is the reduction of the problem. We will introduce several terms to be able to define this part of the algorithm. Let S_i be the suffix starting on the index i and running the sentinel. Suffix S_i is an $S - type$ suffix if it is lexicographically smaller than S_{i+1}. An $L - type$ suffix is a suffix that is larger than the suffix to its right.

We can simply figure out the type of the suffix Si by comparing the character on index i, denoted as S[i], with the character S[i+1]. Si is the S-type suffix if S[i] $<$ S[i+1].

The suffix Si is the L-type if S[i] $>$ S[i+1] holds. But what if S[i] and S[i+1] are the same characters? For example, the suffix starting on index 3 in the string "bioiinformatics" is "iinformatics", and the suffix starting on index 4 is "informatics". Since the first two characters are the same, both are equal "i", we continue the comparison with the right parts of the suffixes, "informatics" and "nformatics". If we take a closer look at them, we can see that they correspond to the suffixes Si+1 and Si+2. So, if we have already compared these two suffixes, then the type of Si is equal to the type Si+1 in the case of equal first characters. The rightmost suffix, sentinel, is defined to be S-type. Considering these facts, it turns out that determining the type of each suffix can be efficiently implemented if we scan the given string from right to left.

To summarize, we will give the rules for determining the suffix type.

- The sentinel is the S-type suffix. The suffix containing only the last character in the string is the L-type suffix since the sentinel is lexicographically smallest character.
- The suffix S_i is the S-type suffix if S[i] $<$ S[i+1], or if S[i] $==$ S[i+1] and Si+1 is the S-type suffix.
- The suffix S_i is the L-type suffix if S[i] $>$ S[i+1], or if S[i] $==$ S[i+1] and Si+1 is the L-type suffix.

The character S[i] is classified according to the type of suffix Si. S[i] is an S-type character if Si is an S-type suffix, and L-type character otherwise.

A character in the given string is said to be the leftmost S character (LMS character) if it is an S-type character that has the L-type character to its immediate left.

3.2. Suffix array search

Once the suffix array is constructed, an O(PlogN) 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, a simple binary search algorithm can find all pattern occurrences.

The algorithm matches nucleotide base by nucleotide base to allow partial matches (pattern's prefix matches). One nucleotide base is matched in each step. Binary search is used to find the left and right index of the range of suffixes with the same nucleotide base (at the position corresponding to the position of the current nucleotide base in

the pattern) as the nucleotide base being matched in that step. In this way, the range of suffixes is reduced at each step. The range of suffixes obtained in the final step represents all occurrences of the pattern in the string.

This way of finding patterns in a string also allows some nucleotide bases to be skipped, which can be very useful if we want to ignore those bases with poor quality. Ignoring nucleotide bases with low-quality represents a great advantage of the suffix array approach over the approaches that work with hash. If pattern appearances are found by comparing hash values, then the pattern must have all nucleotide bases equal to the nucleotide bases in the minimizer sampled from reference; otherwise, their hash values will not be the same. If a sequencing error has occurred in the minimizer extracted from the query, it will not be adequately matched with the minimizers from the reference. It is one of the main reasons why approaches that utilize hash cannot work with large minimizers.

However, when using the suffix array, bases that are most likely to be erroneously sequenced can be skipped, and the sample can be matched, although not all nucleotide bases are equal. We achieve this by not reducing the range of suffixes when the algorithm is in the step that tries to match a base with poor quality. Instead, the algorithm does nothing; it continues the search with the following nucleotide base in the sample.

Ignoring nucleotide bases with poor quality does not help if a deletion occurred. However, in that case, it is still possible to match only the prefix - the part of the sample before the deletion.

Following the example of minimap2 and Winnowmap, HiFiMapper allows setting the algorithm parameters so that the algorithm discards patterns that appear too many times in the reference. If the range of suffixes contains more suffixes than the predefined threshold, none of these matches will be declared an anchor.

Although it contributes significantly to performance, discarding matches often cause fragments that are entirely in repetitive regions not to be mapped. Therefore 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 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.

```
Input suffix_array, query, sample_position, sample_length
Output List of anchors
left_index = 0;
right_index = suffix_array.size;
while (matched_size < sample_length or
 (extended_search and right_index − left_index >
 frequency)) do
    if query.quality(sample_position + matched_size) then
        binary_seacrh_left(left_index, right_index,
         query[sample_position+matched_size]);
        binary_seacrh_right(left_index, right_index,
         query[sample_position+matched_size]);
        if left_index = right_index then
            break;
        end
        matched_size++;
    else
        matched_size++;
        continue;
    end
end
if discard and right_index - left_index > frequency then
    return;
end
if matched_size < sample_size ×min_match then
    return;
end
return anchors;
```

Algorithm 1: Suffix array search

3.3. Chaining

The chaining algorithm implemented in this paper is described in the Minimap2 article. All equations presented in this section are taken from Minimap paper. 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 up to anchor i, denoted as $f(i)$, can be e 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 match size.

Calculating chaining scores with dynamic programming by 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 them h 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 using backtracking, minimap2 identifies the primary chains. The primary chains are chosen not to overlap more than 50% on the query sequence. However, this can cause the discarding of chains with equal chaining scores obtained due to repetitive regions that a given fragment cannot resolve. In the case of several chains of comparable quality, we have decided to report all these chains. However, chains that overlap significantly on the reference will not be printed.

3.4. Longest Common Prefix Array

The longest common prefix array (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. Udi Manber and Gene Myers introduced this auxiliary data structure to speed up the pattern matching algorithm.

The algorithm input is the previously constructed suffix array and the sequence for which it is built. The longest common prefixes should be calculated for adjacent suffixes in sorted order to achieve O(N) time complexity. Kasai's algorithm for LCP construction is used in the HiFiMapper implementation.

If we know that the lcp between two suffixes, denoted as i and j, that are consecutive in the suffix array is the k > 0, we can conclude that the lcp of suffixes i+1 and j+1 is k -1. The suffixes i + 1 and j + 1 are obtained by removing the first letter from the suffixes i and j. Kasai's algorithm iterates through suffixes from longest to shortest to utilize the beforementioned fact and reuse k. However, i + 1 and j + 1 may not be adjacent in the suffix array, so we cannot use the calculated value directly. We know that the suffix i + 1 must be smaller than the suffix j + 1, and there can 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 lcp values stored in the lcp array between these two suffixes.Therefore, all lcp values between the suffixes i + 1 and j + 1 in the lcp array are at least k-1.

Once we have constructed the lcp array, its values can be used to find the leftmost and rightmost nucleotide bases that distinguish the substring from all other substrings in the sequence. A subring (a prefix of a suffix that begins at the position where the substring begins) shares the largest number of common nucleotide bases with its two neighbors in the suffix array. This means that for each substring that starts at position i, the first letter to the right of position i that distinguishes that substring from all other substrings in the sequence corresponds to the maximum between lcp [sa [i]] and lcp [sa [i] -1]. Based on the found rightmost nucleotide, we can also find the leftmost letters that distinguish the substring that ends at position i from all other substrings in the sequence.

Information on the rightmost and leftmost positions that uniquely determine the substring can help us filter out false positive matches that occur due to repetitive regions in the sequence. It is possible to validate each match by matching samples that should be mapped to the leftmost and rightmost positions that distinguish a substring from other substrings. If the validation succeeds, then we keep the right match only.

If validation cannot be performed because the query is not large enough to reach the leftmost and rightmost positions, all matches are preserved.

4. Implementation

The implemented tool allows setting some parameters in order to achieve the best performance for a particular use case. We will briefly describe each parameter below.

```
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>
      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
-r, --repetitive <bool>
  default: false
  creating lcpa and resolving unmapped
-h, --help
  prints the usage
```

The threads parameter allows defining the maximum number of threads that parallelly execute components that can be parallelized. The sample length parameter sets the size of the samples that are extracted from the query and whose exact matches with reference are declared as anchors. The sample count parameter determines the number of samples that are extracted from the query. The min match parameter determines the percentage of the sample that must be exactly matched in order for a match to be declared as an anchor. Setting this parameter to a value less than 1 allows the samples in which the deletion occurred to become anchors. The quality parameter sets a threshold that determines whether the base is of good or poor quality. Setting this parameter allows skipping bases with poor quality while binary searching suffix array. The frequency parameter determines the maximum number of matches between one anchor and the reference. If the discard parameter is set, then all anchors that have more matches with a reference than the specified frequency will be discarded. Discarding anchors can improve execution time, but at the same time, decrease mapping quality. If the extended search parameter is set, then the algorithm tries to reduce the number of matches by mapping the bases outside the sample until the desired number of matches

is reached or when it is no longer possible to expand and match the sample. Setting the repeat parameter allows building lcp array and solving unmapped fragments using the information from the lcp.

4.1. Dependencies

To build the project from the source, you will need Python3, CMake, and C++ compiler.

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.

When time consumption is crucial, another less obvious component can be parallelized - suffix array construction. Even though there exist methods for parallel construction of suffix array, in practice, they have been shown to improve run times only three to four times since there is a lot of required synchronization. However, if the reference is divided into disjoint parts, then a separate suffix array can be constructed for each part, and all these suffix arrays can be built parallelly. In conducted experiments, this approach has been shown to contribute to the time complexity significantly.

When time consumption is crucial, another less obvious component can be parallelized - suffix array construction. Even though there exist methods for parallel construction of suffix array, in practice, they have been shown to improve run times only three

to four times since there is a lot of required synchronization. However, if the reference is divided into disjoint parts, then a separate suffix array can be constructed for each part, and all these suffix arrays can be built parallelly. In conducted experiments, this approach has been shown to contribute to the time complexity significantly. Nevertheless, this approach should be used with caution because the search can be performed more efficiently in one long sequence than in many small sequences. For example, if we have 1×10^7 long sequence, then the basic search that takes O(Plogn) time is about ten times faster than performing ten O(Plog(n/10)) searches because the logarithm function grows slowly. To conclude, if the preprocessing time is not crucial, it is better to construct larger suffix arrays. Still, if the construction time is important and very few queries are processed, it is better to split the reference.

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 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. The only requirement in this version is the python package tabulate. The library allows printing tables in several formats, including latex tables. It is used in the benchmark framework to enhance the testing process.

5. Results and Discussion

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, minimap2. The framework comes with a simple API for specifying the 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 reads generated by the HiFi simulator.

The mapping of the read is 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 read 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 reads since there is

no ground truth for real reads. Therefore, in the experiments including real data, the evaluation metric is the ratio of mapped reads and the total number of reads.

Each experiment shown in this chapter was conducted five times, and mean values of metrics were calculated. Additionally, in all experiments, including time measurement, the measuring unit is second.

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	256
Memory:	738 GiB

5.2. Map

5.2.1. Artificial Data

We evaluated the influence of different parameters on the performance of HiFiMapper. Experiments in this section were performed on completely artificially generated data to quickly conclude how parameters affect quality 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 described in the previous section. 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 in the suffix array is $O(plogn)$, 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. This experiment also confirms the correctness of the tool implementation since all fragments are accurately mapped.

l / c	10	20	50	70	100
25	1.0, 0, 1.27	1.0, 0, 1.77	1.0, 0, 3.76	1.0, 0, 4.99	1.0, 0, 7.32
50	1.0, 0, 1.32	1.0, 0, 1.87	1.0, 0, 4.03	1.0, 0, 5.16	1.0, 0, 7.59
75	1.0, 0, 1.22	1.0, 0, 1.93	1.0, 0, 4.07	1.0, 0, 5.54	1.0, 0, 8.04
100	1.0, 0, 1.39	1.0, 0, 1.86	1.0, 0, 2.99	1.0, 0, 5.89	1.0, 0, 8.18
150	1.0, 0, 1.52	1.0, 0, 2.18	1.0, 0, 5.01	1.0, 0, 6.28	1.0, 0, 9.12

Tablica 5.1: The influence of sample length and sample count parameters on performance.

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 are generated so that all bases are the same within all 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. In this experiment, the parameters were set to discard samples that have more than 10 matches.

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 repeating segments. With an insufficient amount of information due to low coverage and discarding of the samples, the fragments that should be mapped to some repetitive segment are mapped to several repetitive segments causing poor quality when we evaluate all the mappings found by the tool.

Minimap2 and Winnowmap were run on the same data set. Minimap2 and Winnowmap were run on the same data set. Both tools achieved the same mapping quality of 0.978.

Minimap2 was run with:

```
minimap2 -x map-hifi -t 256 reference.fasta
    fragments.fastq
```

Winnowmap was run with:

l / c	10	20	50	70	100
25	0.868, 140, 1.19	0.917, 0, 1.21	0.992, 0, 3.11	0.998, 0, 4.31	1.0, 0, 3.94
50	0.939, 25, 1.15	0.99, 0, 1.76	1.0, 0, 3.41	1.0, 0, 4.48	1.0, 0, 6.26
75	0.977, 14, 1.19	0.999, 0, 1.37	1.0, 0, 2.56	1.0, 0, 5.13	1.0, 0, 6.66
100	0.992, 3, 1.11	1.0, 0, 1.85	1.0, 0, 3.81	1.0, 0, 5.18	1.0, 0, 6.9
150	0.999, 0, 1.26	1.0, 0, 1.71	1.0, 0, 3.98	1.0, 0, 5.57	1.0, 0, 7.72

Tablica 5.2: The influence of sample length and sample count parameters on performance when mapping against highly repetitive reference.

```
winnowmap -W repetitive_k15.txt -x map-pb -t 256
    referece.fasta fragments.fastq
```

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. In the previous experiment, the algorithm found many wrong mappings in addition to the correct mappings or was unable to find the accurate mapping because all anchors were discarded. We expect extended search heuristics to find the key differences that determine the correct position.

From the data shown in the Table 5.3, we can conclude that the extended search heuristic positively contributes to mapping accuracy without significantly affecting the mapping time.

l / c	10	20	50	70	100
25	1.0, 0.0, 1.08	1.0, 0.0, 1.7	1.0, 0.0, 3.33	1.0, 0.0, 4.73	1.0, 0.0, 6.39
50	1.0, 0.0, 1.07	1.0, 0.0, 1.67	1.0, 0.0, 3.48	1.0, 0.0, 4.94	1.0, 0.0, 6.86
75	1.0, 0.0, 1.34	1.0, 0.0, 1.74	1.0, 0.0, 3.59	1.0, 0.0, 5.01	1.0, 0.0, 6.96
100	1.0, 0.0, 1.2	1.0, 0.0, 1.88	1.0, 0.0, 3.91	1.0, 0.0, 5.3	1.0, 0.0, 7.13
150	1.0, 0.0, 1.23	1.0, 0.0, 1.99	1.0, 0.0, 4.24	1.0, 0.0, 5.88	1.0, 0.0, 8.12

Tablica 5.3: The influence of sample length and sample count parameters on performance when mapping against highly repetitive reference.

5.2.2. Simulated Data

In this section, we will evaluate HiFiMapper on datasets including real references and reads that are simulated from these references using the described HiFi simulator.

Homo sapiens chromosome 13 is a real reference used in this section. Experiments were performed on this reference since it contains many repetitions and usually presents challenge for aligning tasks.

We will first investigate how the length and number of samples affect the mapping quality, the number of unmapped fragments, and the mapping time. To begin with, we conducted an experiment with the discarding of samples that match more than the given frequency because in the presence of many repetitive regions, the time of chaining samples becomes the bottleneck. For this experiment, 1×10^5 fragments of size 25000 were generated with a sequencing error probability of 0.001. Samples in queries are selected by the sequential method.

An experiment was performed on the same data set as in the previous experiment to verify how different sample length and frequency values affect mapping accuracy, the number of unmapped samples, and mapping time. The data in the table confirm the assumption that discarding samples increases the number of unmapped fragments because if the fragment is whole in a repeating region, there is a high possibility that all its samples will be discarded.

l / f	10	50	100
50	0.982, 1010, 48.81	0.978, 628, 60.97	1.0, 2, 123.59
100	1.0, 2851, 28.1	1.0, 2820, 30.54	1.0, 9, 36.83
200	1.0, 2867, 16.54	1.0, 2830, 16.2	1.0, 4, 19.71

Tablica 5.4: The influence of sample length and frequency parameters on performance when mapping against human chromosome 13.

5.2.3. Real Data

6. Conclusion

Zaključak.

LITERATURA

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.

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.

Fast Overlapping Single Molecule Highly Accurate Sequencing Data

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.