UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 291

RAPID ALIGNMENT OF HIGH-FIDELITY SEQUENCING DATA

Mauro Staver

Zagreb, June 2021

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 291

RAPID ALIGNMENT OF HIGH-FIDELITY SEQUENCING DATA

Mauro Staver

Zagreb, June 2021

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 12 March 2021

BACHELOR THESIS ASSIGNMENT No. 291

Student:	**Mauro Staver (0036514400)**
Study:	Electrical Engineering and Information Technology and Computing
Module:	Computing
Mentor:	prof. Mile Šikić

Title: **Rapid Alignment of High-Fidelity Sequencing Data**

Description:

Third-generation sequencing technologies facilitate de novo assembly due to longer sequenced fragments comparing with previous technologies. Recently, Pacific Biosciences has improved used chemistry and sequencing device performances, resulting in a new sequencing protocol for the production of high fidelity reads. The probability distribution of read length is narrow, with the mean value at 25k without fat tails. Yet, the accuracy of more than 99% is even more impressive. These high accuracy data enables the creation of algorithms less sensitive to error rates. In addition, the high accuracy allows the reduction of the running time required for mapping reads to a reference genome. The main aim of this thesis is an adaptation of publicly available algorithms for the alignment of long error-prone reads or the invention and implementation of a new algorithm that will be tailored to recent high-quality reads. The solution should be appropriated 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: 11 June 2021

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 291

Pristupnik:	**Mauro Staver (0036514400)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Mile Šikić

Zadatak: **Brzo poravnanje visoko pouzdanih dugačkih očitanja**

Opis zadatka:

Tehnologije treće generacije uređaja za očitanje značajno olakšavaju problem sastavljanja genoma zbog mogućnosti očitanja znatno duljih fragmenata od prethodnika. Nedavno, tvrtka Pacific Biosciences, je poboljšala korištene kemikalije i performanse uređaja za očitanje što je rezultiralo novim protokolom koji proizvodi visoko pouzdane fragmente. Distribucija duljine fragmenata ovoga protokola je oko 25 kbp bez teških repova, no najimpresivnija je točnost od iznad 99%. Činjenica da podaci imaju visoku točnost može se koristiti za osmišljavanje algoritma manje osjetljivih na pogrešku. Visoka točnost omogućuje smanjenje vremena potrebnog za poravnanja fragmenata i referentnog genoma. Glavni cilj ove teze je prilagodba javno dostupnih algoritama za poravnanje 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 jednu od OSI odabranih licenci.

Rok za predaju rada: 11. lipnja 2021.

Table of contents

Introduction

Mapping DNA reads to a reference genome is a common problem in bioinformatics. It is usually much less computationally expensive than assembling reads into a genome or transcriptome. There are many mapping tools that implement different approaches that have evolved from the seed-and-extend approach to newly popular sketch-based methods. Sketch-based algorithms achieve fast runtimes on large sequences because they employ dimensionality reduction, using only a subset of k-mers present in the sequence to compactly represent the whole sequence. Minimap2 is a versatile mapping/alignment tool that implements this idea and is considered the industry standard.

Advancements in DNA sequencing technologies have led to an explosive growth of the amount of sequencing data being generated every year. As such, efficient algorithms are needed in order to process this large amount of data. The three major sequencing technologies are Illumina, Pacific Biosciences SMRT and Oxford Nanopore Technologies. As these technologies become cheaper and produce more accurate and longer reads, algorithms can be improved by utilizing the properties of technology specific reads.

Usually there is a tradeoff between read accuracy and length, so algorithms were developed for both short and highly accurate reads and long but error-prone reads. However, the latest advancement in Pacific Biosciences SMRT technology produces both highly accurate and long reads called High Fidelity (HiFi) reads.

In this paper we propose a new mapping algorithm, hifimap, which utilizes properties of HiFi reads in order to achieve faster runtimes and lower memory requirements without sacrificing mapping accuracy. We will introduce the hifimap algorithm, discuss its implementation and evaluate the performance and accuracy of hifimap by comparing it to minimap2 on both simulated and real data.

1. Methods

HiFi reads have an accuracy higher than 99% and a light-tailed narrow length distribution with the maximal mean of around 25kbp. Hifimap uses these properties in order to identify short candidate regions on the reference genome and then tries to find a mapping in those regions by finding minimizer (Roberts et al., 2004) matches and chaining them in a similar manner as minimap2. The idea is to minimize the length on which chaining is performed. Finding candidate regions is done by counting the number of minimizer hits in each region and using a strategy to choose the most fitting candidates. Furthermore, mapping positions can be approximated with a slightly lower accuracy just by observing minimizer hits in neighbouring regions without the need for chaining.

1.1. Overlapping Regions

We partition the target sequence into logical regions of the same length where each two neighbouring regions overlap by exactly half region length.

Definition

Region R_r of some target sequence is an interval of positions $[rA, (r+2)A>$, $r = 0, \ldots, floor(L/A) - 1$, where A = half region length and L = target sequence length.

A target sequence contains exactly $floor(L/A)$ regions, where each region R_r starts at position rA and has length $2A$, as depicted in Fig 1.1.

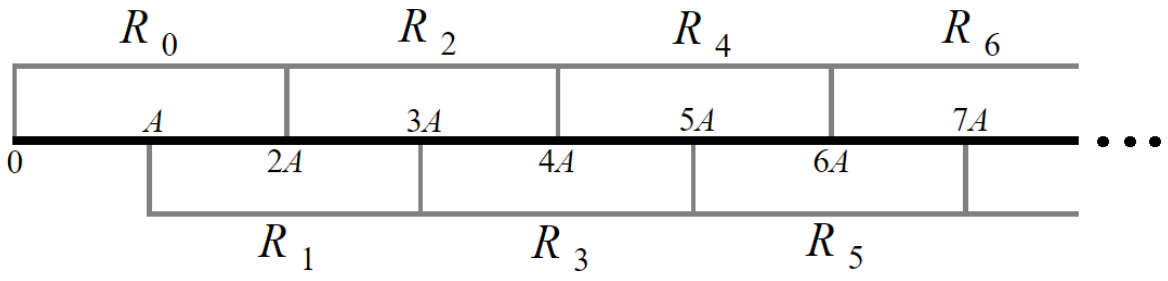

Fig 1.1 Partitioning target sequence into regions

Lemma

If a substring of length A is taken from the target sequence, its starting and ending positions must both be in at least 1 same region.

In other words, at least 1 region will fully contain the taken substring.

Proof

Let's assume a target sequence of length LA.

Let I_r be an interval of target sequence positions $[rA, (r+1)A >$, $r = 0, \ldots, L-1$.

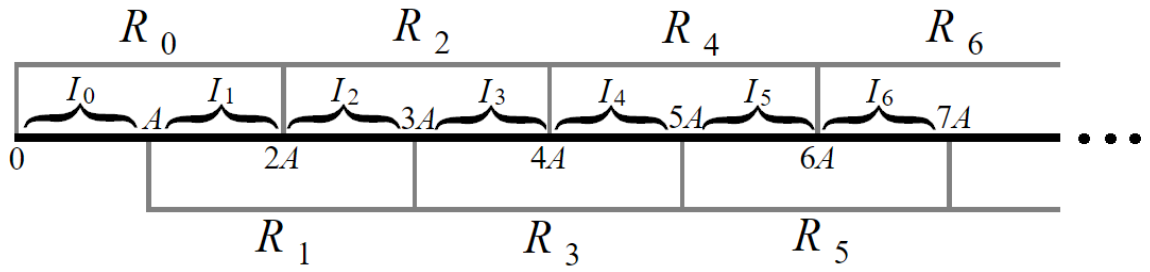

Fig 1.2 Intervals I_r on target sequence

Then the following axioms hold:

1. region R_r consists of exactly positions $I_r \cup I_{r+1}$

2. if a substring of length A has a starting position in I_r, then it's ending position must be in I_{r+1}.

From (1) and (2) we can infer that all substrings of length A with a starting position in I_r are fully contained in region R_r.

Since intervals $I_0, I_1, \ldots, I_{L-1}$ contain all possible starting positions of substrings of length A, the lemma must hold.

Generally, the reference sequence length doesn't have to be a factor of A, but it's not difficult to show that the same idea still applies. Notice that all positions from

$[A, (L-1)A >$ belong within 2 regions, while the first A and last A positions of the reference sequence belong to only 1 region.

In the context of mapping, substrings of length A taken from the target sequence are HiFi reads. As mentioned, HiFi reads have a narrow length distribution so we can set A to be the average read length in our dataset and as a result, a significant percentage of each read will be contained inside 1 region. In practice we usually set A to be a bit higher than the average read length since we want each read to be fully covered by 1 region. Hifimap algorithm is quite robust to large values of A, however setting A too high will affect hifimaps performance. The larger we set A, the more hifimap algorithm morphs into the minimap algorithm. If we set A to equal half the target sequence length, hifimap algorithm and minimap algorithm become equivalent. Our approximation method on the other hand is quite sensitive to values of A as it only performs well when A ~ read length.

1.2. Minimizer Index

Minimizer sampling was introduced by Roberts et al. (2004). It is a method of selecting k-mers from a sequence such that if two sequences are similar enough, the same k-mers will be selected from both. Minimap2 collects reference sequence minimizers and indexes them in a hash table with the key being the minimizer value, and value being a list of positions of minimizers whose value is the same as the key. It then maps each query sequence by sampling its minimizers and querying the index in order to find exact matches. These matches are then chained using dynamic programming and exact mapping positions are found.

The idea behind the hifimap algorithm is to rapidly count the number of minimizer hits in each region and then compute and chain the exact minimizer matches only on most prominent regions. In order to achieve this, hifimap employs a similar indexing scheme as minimap2 but instead of mapping each minimizer value to a list of positions, it maps each minimizer value to a list of regions which contain that minimizer. Alongside each region, it also stores term frequency, the number of times a specific minimizer has occurred in that region. We utilize this value in the

counting procedure in order to not overcount the number of minimizer hits in each region.

Since we don't store each position of the minimizer's occurrence but merely it's regions, we expect our index to be more memory efficient. In particular, we achieve memory savings when the same minimizer occurs multiple times in the same region.

1.2.1. TFIDF Index Filtering

Minimap2 dismisses the most frequent reference minimizers in order to reduce noise in the mapping procedure. Hifimap's region partitioning allows us to employ the TFIDF(Term frequency - Inverse document frequency) method from Natural Language Processing in order to achieve the same goal. TFIDF is widely used by search engines to determine how significant a keyword is to a document in a collection or corpus. Similarly, hifimap uses TFIDF to determine the significance of minimizer occurrences in each region within a context of a corpus of all existing regions on the target sequence.

We can denote each index entry as a 3-tuple $e = (m, R_r, tf)$ where m is the minimizer value, R_r the region and tf the term frequency of minimizer m in region R_r.

We compute the TFIDF score for each index entry e with the following formula:

$$TFIDF(e) = TF(e) * IDF(e) \quad (1)$$

where

$TF(e) = e.tf$

$IDF(e) = log_e\left(\frac{total\ number\ of\ regions}{number\ of\ distinct\ regions\ minimizer\ e.m\ appears\ in}\right)$

$TF(e)$ represents the minimizer's importance in a specific region and $IDF(e)$ represents the minimizer's cross-region importance. Multiplying these two terms yields a value that represents the minimizer's importance in a specific region in the context of all existing regions.

We expect that each region contains approximately the same number of minimizers, so there is no need to normalize the term frequency by dividing it with the total number of (not necessarily distinct) minimizers that appear in the observed region, as is usually done in the TFIDF method when documents contain differing amounts of keywords.

Before inserting a 3-tuple e in the Index, we compute it's TFIDF value and compare it to the threshold value. If $TFIDF(e)$ passes the threshold, we insert it in the Index, otherwise we dismiss it. This further reduces Index memory requirements, but adds additional time overhead to Index construction since we need to compute the minimizer term frequency and count the number of regions minimizer $e.m$ appears in, in order to compute $TFIDF(e)$.

It is not clear how to choose the appropriate TFIDF threshold. Hifimap sets the threshold to be the TFIDF value of singleton minimizers, minimizers with values that appear only once in the minimizer sample.

$$TFIDF_{threshold} = log_e\left(\frac{total\ number\ of\ regions}{2}\right) \quad (2)$$

This way we are certain that at least singleton minimizers will pass the TFIDF filter which is important since matching singleton minimizers is the best indication of an appropriate mapping.

In our implementation we allowed users to offset this value by some constant defined at the start of the program in order to allow further experimentation.

1.2.2. Index Construction

Hifimap's Index is constructed by first collecting reference minimizers and sorting them by value. We can then construct the Index with one iteration over the sorted list of minimizers by buffering regions of minimizers with the same value, since they will appear as neighbours in the sorted minimizer list. Each buffer counts the number of occurrences of each inserted region and hence contains (region, term frequency) pairs of all minimizers with the same specific value. Each buffer contains enough information to compute the TFIDF of all pairs inside the buffer. When there are no more regions to be inserted in the current buffer, we can iterate

over the buffer and compute TFIDF scores of all pairs and if they pass the threshold insert them in the Index.

Algorithm 1 Index construction

Input: Target sequence

Output: Minimizer hash table

```
Function ConstructIndex(target):
Sketch = ComputeMinimizers(target)
SortByValue(Sketch)
Index ← Empty hash table
Buffer ← Empty buffer object
i ← 1
while i < Sketch.size():
      if Sketch[i] == Sketch[i-1]
            Buffer.insert(Sketch[i-1].regions())
      else
            Buffer.insert(Sketch[i-1].regions())
            CheckThresholdAndInsert(Buffer, Index)
            Buffer.clear()
      i++
return Index
```

In the given pseudocode `buffer` is an object that counts the number of occurrences of each region (ie. term frequency) and also stores the number of distinct regions inserted in the buffer (used to compute Idf). `CheckThresholdAndInsert()` is a method that computes TFIDF values for each (region, term frequency) pair in the buffer and if it passes the TFIDF threshold, inserts the pair in the Index under the key of the minimizer value represented by the buffer. `ComputeMinimizers()` simply returns a minimizer sample from the given sequence.

1.3. Mapping

Mapping a query sequence to the target sequence means identifying intervals on the target which are highly similar to the corresponding intervals on the query. The result of mapping can be 0 or more found intervals. Since target sequences can be very large, hifimap first reduces the search space by identifying regions which seem the most prominent to contain some highly similar interval. Then it finds and chains exact minimizer matches between the query sequence and selected regions. The positions of the first match in every chain correspond to the appropriate starting positions of significantly similar intervals between the query and target sequence. In the same manner, positions of the last match in every chain correspond to the ending positions of the same intervals.

1.3.1. Counting Hits

Hifimap first collects minimizers from the query which are then used as seeds to query the Index. Given that minimizer m occurs k times in the query, then for each (R_r, tf) pair obtained from querying the Index using the key m we apply the following heuristic to count region hits:

$$Hits[R_r] \mathrel{+}= min(k, tf) \tag{3}$$

Notice that after applying this heuristic to every distinct query minimizer, $Hits[R_r]$ gives us a tight upper bound on the length of a chain consisting of minimizer matches between the query and region R_r. Our intuition is simple: the higher the upper bound on the length of a chain, the higher the probability that a significantly similar interval exists between region R_r and the query. Conversely, a low upper bound on the length of a chain strictly limits the significance of a possible similar interval between the region and the query. This is a heuristic since there is no guarantee that an actual chain of length $Hits[R_r]$ will be found because that depends on the relative ordering of minimizer matches which we didn't take into account. Nevertheless, this does allow us to efficiently identify regions where a long chain is possible, and dismiss regions where it is not.

Another heuristic that imposes naturally would be $Hits[R_r] \mathrel{+}= k \cdot tf$ since there are $k \cdot tf$ exact minimizer matches with the minimizer value m between the query and region R_r. This heuristic however is too optimistic and could lead to overcounting hits and thus overestimating the likelihood of finding a significantly similar interval in the given region.

Example:

Assume that minimizer m_k appears k times in the query and $k = 1, \ldots, 5$.

For simplicity, assume Index:

$$m_1 \rightarrow [(R_7, 10)]$$

$$m_2 \rightarrow [(R_9, 3), (R_{12}, 3)]$$

$$m_3 \rightarrow [(R_5, 1), (R_9, 3)]$$

Recall that our Index is a hash table in which each minimizer is mapped to a list of (R_r, tf) pairs, meaning that the given minimizer appears tf times in region R_r.

Applying heuristic (3) would yield:

$$Hits[R_5] = 1,\ Hits[R_7] = 1,\ Hits[R_9] = 2 + 3 = 5,\ Hits[R_{12}] = 2$$

Take for example region R_9, minimizer m_2 appears 2 times in the query sequence and 3 times in R_9, so we add $min(2, 3)$ to $Hits[R_9]$. In the same way, minimizer m_3 appears 3 times in the query sequence and 3 times in R_9, so we add $min(3, 3)$ to $Hits[R_9]$.

By observing these Hits values we can conclude that the longest chain is possible in region R_9 and therefore R_9 is the most prominent region.

On the other hand applying the heuristic $Hits[R_r] \mathrel{+}= k \cdot tf$ would yield:

$$Hits[R_5] = 3,\ Hits[R_7] = 10,\ Hits[R_9] = 6 + 9 = 15,\ Hits[R_{12}] = 6$$

We would again identify region R_9 as the most prominent region, however it is hard to provide a sensible interpretation of these results. Notice also that region R_7 is drastically overvalued.

We will now supply the pseudocode for counting hits.

Algorithm 2 Counting hits

Input: Query minimizers sorted by value, Target Index

Output: Hits array

```
Function CountHits(QuerySketch, Index):
Hits[] ← Array of size = total number of regions, all elements set to zero
i ← 1
cnt ← 1
while i < QuerySketch.size():
      if QuerySketch[i] == QuerySketch[i-1]
            cnt++
      else
            for each (Rr, tf) in Index[QuerySketch[i-1]]:
                  Hits[Rr] += min(cnt, tf)
            cnt ← 1

return Hits
```

The time complexity of this algorithm is $O(Q * L/A)$, where Q is the length of the query sequence, L the length of the target sequence and A half of the region length. The term L/A, after flooring, equals the total number of regions in the reference sequence. We expect this algorithm to be somewhat faster than minimap2's algorithm for finding minimizer matches between the query and target sequence since the total number of regions is less than the total number of possible minimizer positions.

1.3.2. Selecting Regions

After computing Hits, hifimap chooses the most prominent regions on which it will find and chain exact minimizer matches. The simplest strategy is to select k regions with the highest Hits score that is larger than some threshold h, where k and h are user defined constants. This strategy is inefficient as it overlooks the fact that the regions are overlapping so in practice it often chooses neighbouring regions which is redundant. It's important to note that each additional selected region adds additional overhead which we want to minimize. Let's suppose that a significant mapping can be found in region R_r. Then it is highly probable that regions R_{r-1} and R_{r+1} will both have a large Hits score as they overlap with region R_r. Furthermore, the following invariant holds:

$$Hits[R_r] \leq Hits[R_{r-1}] + Hits[R_{r+1}],\ r = 1, \ldots,\ floor(L/A) - 1 \qquad (4)$$

We will not formally prove this invariant but we will provide some intuition. All hits in the first half of region R_r are also hits in region R_{r-1} and all hits in the second half of region R_r are also hits in region R_{r+1}. However, hits contained in the first half and the second half of regions R_{r-1} and R_{r+1} respectively, are not part of region R_r and hence the less or equal sign.

If a region R_r has a higher Hits score than each of its neighbours, there is no need to select the neighbours. Selecting only region R_r is sufficient since it contains most, if not all hits from both its neighbouring regions. Therefore we propose the following strategy of selecting the most prominent regions:

Select k regions with the highest Hits score that is larger than some threshold h without selecting neighbouring regions. (5)

In practice, a low value for k yields accurate and satisfying results and minimizes overhead.

1.3.3. Approximating Mapping Positions

After computing $Hits$, ie. upper bounds on the lengths of chains, we can already approximate mapping positions on the reference just by observing Hits scores of neighbouring regions. Consider the following example:

Suppose a substring of length A was taken from the reference and $S = Hits[R_r] = Hits[R_{r-1}]$ is the highest Hits score. $Hits[R_r] = Hits[R_{r-1}]$ gives us a strong indication that the substring is taken from the interval $[rA, (r+1)A >$ as depicted in Fig 1.3. We can not guarantee that the substring is taken from the mentioned interval because of noise that might be originating from the first half and second half of regions R_{r-1} and R_r respectively, however in practice this is highly unlikely.

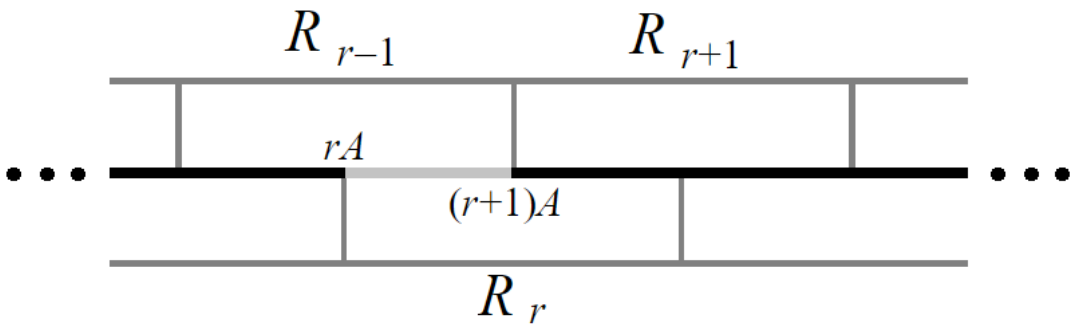

Fig 1.3 Substring taken from an interval colored light gray

We can further generalize this by observing the ratio between Hits scores of neighbouring regions. If for example $Hits[R_{r-1}]$ makes up 60% of $Hits[R_r]$and $Hits[R_{r+1}]$ makes up the remaining 40% we can approximate the mapping interval as: $[(r+1)A - 0.6A, (r+1)A, + 0.4A >$, as depicted in Fig 1.4.

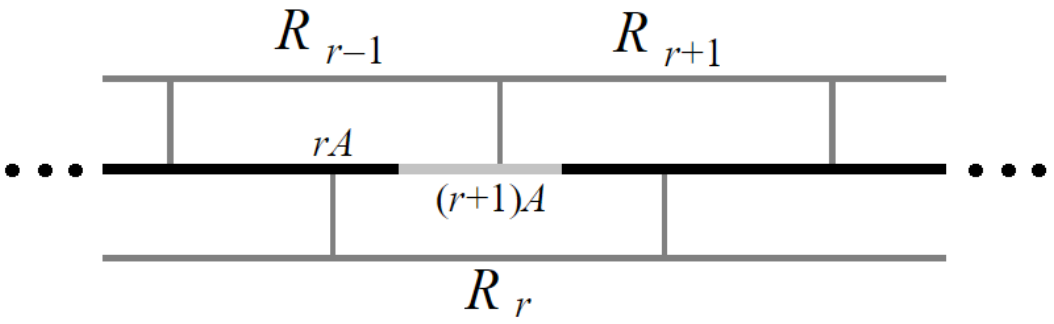

Fig 1.4 Substring taken from an interval colored light gray

The general conclusion is as follows: If region R_r is a selected region, we can approximate the mapping interval on the target sequence as:

$$\left[(r+1)A - \frac{Hits[R_{r-1}]}{Hits[R_r]} \cdot A,\ (r+1)A + \frac{Hits[R_{r+1}]}{Hits[R_r]} \cdot A\right>$$

and we simplify it to:

$$[p,\ p + A >, \text{where } p = (r+1)A - \frac{Hits[R_{r-1}]}{Hits[R_r]} \cdot A$$

For the special case when region R_0 is selected, we can approximate the interval as:

$$[p - A,\ p >, \text{where } p = (r+1)A + \frac{Hits[R_1]}{Hits[R_0]} \cdot A$$

This approximation method implies that the entire query sequence is mapped to the approximated interval. In terms of accuracy, it relies heavily on the assumption that A ~ query sequence length. We have also found that the density of minimizers can affect the accuracy, since higher density leads to more precise Hits ratios.

1.3.4. Finding Matches

For each selected region R_r hifimap finds exact minimizer matches between the query sequence and region R_r. This can be accomplished in a few different ways.

If a large k value is expected, or in other words, we need to find minimizer matches between the query sequence and a large number of regions, perhaps the most efficient approach is to index query minimizers in a minimap2-like fashion. Then we can simply iterate over minimizers of each selected region and find matches by querying this index. On the other hand, indexing proves to add a significant overhead for low values of k which are used in practice. Therefore hifimap adopts a different approach.

For each selected region hifimap collects region minimizers and sorts them by value and then employs a 2-pointer algorithm similar to the merge subroutine in merge-sort in order to find minimizer matches. Keep in mind that we have previously collected and sorted query minimizers in order to count hits. This algorithm is commonly used in read-to-read mapping tools and makes sense since we can consider each region as a target read which is twice longer than the query read.

We will now address different approaches to collecting region minimizers. The most straightforward approach is to simply compute the minimizers dynamically by iterating over each selected region. Considering all k regions, the time complexity of this is $O(kA)$ but in practice we can consider k and A to be constants so this would effectively be $O(1)$. In fact, we can consider the entire process of finding minimizer matches to be $O(1)$ in practice. This constant time, however, is quite large and noticeable as it consumes a considerable share of runtime in the entire mapping procedure when mapping to short reference sequences. For example, in our experimentation dynamically computing region minimizers took up ~60% of runtime of the entire mapping procedure when mapping to a target sequence of length 10^6. As expected, we have found that this percentage decreases as we increase the target sequence length.

It's important to notice that we have already computed minimizers from all regions in the Index Construction algorithm but we didn't store them in memory. In order to

make hifimap competitive with other mappers on both long and short reference sequences, we decided to store precomputed minimizers from all regions in memory and then we can simply fetch minimizers from selected regions at runtime. This of course comes with an additional memory cost, however it is used only when mapping to short reference sequences which don't take up much memory anyway. On large reference sequences, hifimap sticks with dynamically computing minimizers as this constant time becomes imperceptible and the additional memory cost would be too great.

If we don't have strict memory requirements and using more memory for short reference sequences is acceptable, the process of finding matches can be further improved by precomputing minimap2-like indexes for each existing region. The algorithm would then simplify to just iterating over query minimizers and finding matches by querying indexes of selected regions. Hifimap doesn't implement this approach for the sake of simplicity and since bioinformaticians mostly care about large reference sequences.

1.3.5. Chaining

After finding minimizer matches between the query sequence and all selected regions, the last step is to perform chaining. Chaining is a process of finding the maximal collinear subset of matches by solving the longest increasing subsequence problem. Before actual chaining, clustering is performed on the list of found matches and a chain is computed for each cluster. Each chain then represents a mapping result.

Hifimap performs chaining in the same fashion as minimap2. However, the list of found matches on which chaining is performed is significantly smaller since it consists only of matches found on selected regions. On the other hand, minimap2 performs chaining on matches found throughout the whole reference sequence. We argue that this difference will be noticed in runtimes when mapping to sequences with highly repeating minimizers since minimap2 should then find a significantly larger number of minimizer matches that need to be chained.

1.3.6. Pseudocode and Analysis

At the end of this section, we provide the pseudocode and analysis of the entire mapping procedure.

Algorithm 3 Mapping a query sequence

Input: Query sequence, Target Index

Output: Chained minimizer matches

```
Function Map(Query, Index, k, h):
QuerySketch = ComputeMinimizers(Query)
SortByValue(QuerySketch)
Hits = CountHits(QuerySketch, Index)
Matches ← Empty list
for each region in SelectRegions(Hits, k, h):
      RegionSketch = CollectRegionMinimizers(region)
      for each match in FindMatches(QuerySketch, RegionSketch):
            Matches.insert(match)
return Chain(Matches)
```

In the given pseudocode `SelectRegions(Hits, k)` is a function that returns a list of selected regions using some strategy, for example strategy (5).

`CollectRegionMinimizers(region)` returns a list of minimizers belonging to a given region. It can be implemented in a few different ways as discussed in section 1.3.4. `FindMatches(QuerySketch, RegionSketch)` returns a list of minimizer matches found between the two given sketch lists using the 2-pointer algorithm mentioned in section 1.3.4 or some other method. The pseudocode for `CountHits(QuerySketch, Index)` is given in section 1.3.1.

In our analysis we will use the following notation:
Let L be the length of the target sequence, Q the length of the query sequence and A half of the region length.

In terms of time complexity, notice that only functions `CountHits()` and `SelectRegions()` depend on the length of the target sequence. This dependency is further dampened because they do not solely depend on L, but actually depend on the total number of regions which is A times smaller than L. Their time complexities are $O(Q \cdot L/A)$ and $O(L/A)$, respectively, assuming $k = 1$ is given to `SelectRegions()` so it only needs to find the region with the maximum $Hits$ score. We can consider everything else to be constant time overhead, since Q ~ A ~ const. in practice. Also, it's worth mentioning that the worst case scenario for `CountHits()` is highly unlikely to occur.

On the other hand, minimap2 finds and chains exact minimizer matches between the query and the entire reference sequence. We argue that the time complexity of this has a larger multiplicative constant hidden in its Big O notation than that of hifimap. Therefore, we expect that asymptotically, the runtime of hifimap should grow slower than the runtime of minimap2.

We have tested hifimap on both synthetic and real data and compared the results to minimap2. The accuracy was estimated as a percentage of minimap2 mappings which have equivalent hifimap mappings with some tolerance of t base pairs.

2. Implementation

Hifimap is implemented in C++ and supported on Linux, MS Windows, and Mac OS. It can be used as a C++ library or as a stand-alone application. We have implemented hifimap on top of ram, which is a C++ implementation of minimap with a few modifications.

Hifimap offers quite a large number of options and can be widely configured.

Hifimap's dependencies are:

- gcc 4.8+ | clang 3.5+
- cmake 3.11+
- pthread
- zlib 1.2.8+
- rvaser/biosoup 0.10.0
- rvaser/thread_pool 3.0.3
- rvaser/bioparser 3.0.13

Hifimap uses the module bioparser for parsing input sequences, biosoup for efficient data structures used to store biological data and thread_pool for mapping query sequences in parallel on multiple threads.

2.1. Details

When creating the Index, we do not directly insert (region, term frequency) pairs in the hash table but rather append them in an array. The hash table only keeps the intervals on the array. This is designed to reduce heap allocations and cache misses.

We have implemented strategy (5) as a region selection strategy and have optimized it for $k = 1$ to only find the region with the maximum Hits score.

An approximation method described in section 1.3.3. is used when given the option `--approx`. This results in a faster runtime as we only compute Hits and immediately approximate mapping positions but may reduce accuracy.

Following the discussion on collecting region minimizers in section 1.3.4, we have given the user the option to choose between extra time overhead or additional memory cost. By default hifimap uses additional memory in order to reduce the time overhead. This can and should be toggled when mapping to large reference sequences using the option `--save-mem` (for the lack of a better name).

If given the option `--minhash`, hifimap uses only a portion of collected minimizers which leads to significant runtime improvement but could affect accuracy.

All of the sorting in hifimap is done using Radix sort.

3. Results

Hifimap was able to achieve competitive runtimes on both short and long reference sequences, while especially excelling on very large sequences with highly repeating minimizers. The constant time overhead proved to be significant when mapping to short sequences with a large percentage of singleton minimizers. However, it seems to pay off when reference sequence length increases.

We have conducted tests on both minimap2 and hifimap and compared the results. Accuracy of hifimap was estimated relative to minimap2, by computing a percentage of minimap2 mappings for which an equivalent hifimap mapping exists with tolerance of t base pairs. We denote this percentage as *coverage*, in order to not confuse it with actual mapping accuracy.

In order to simulate HiFi reads, we have synthetically generated for each reference sequence 100k reads, all of length exactly 20k base pairs, with an error rate of 0.5%. At the end we also present results on an actual HiFi dataset.

All tests were conducted using 4 threads with kmer length set to 15 and window length set to 5, all other options were left as default on both minimap2 and hifimap. We will now present results from our experimentation with differing reference sequence lengths.

Table 3.1 Synthetic target, length 10^6, 99.94% singleton minimizers

	minimap2	hifimap	hifimap --save-mem	hifimap --approx
Indexing time	0.078s	0.2676s	0.21s	0.2s
Mapping time	128.3s	182.4s	227.7s	136.9s
Peak RSS	0.964GB	0.34GB	0.33GB	0.33GB
# mappings	100000	100000	100000	100000
coverage t = 30bp	-	100%	100%	33.6%
coverage t = 100bp	-	100%	100%	82.02%
coverage t = 200bp	-	100%	100%	98.78%

Table 3.2 Synthetic target, length 10^8, 93.59% singleton minimizers

	minimap2	hifimap	hifimap --save-mem	hifimap --approx
Indexing time	7.63s	15.76s	13.67s	12.41s
Mapping time	170.1s	232.7s	282.3	206.2s
Peak RSS	2.23GB	2.97GB	2.43GB	2.43GB
# mappings	100000	100000	100000	100000
coverage t = 30bp	-	100%	100%	30.5%
coverage t = 100bp	-	100%	100%	77.55%
coverage t = 200bp	-	100%	100%	97.64%

It seems that, compared to minimap2 hifimap performs worse on short reference sequences. We would argue that a high percentage of singleton minimizers is an even more important factor than the reference sequence length. Due to most minimizers being singletons, minimap2 finds a low number of minimizer matches and does so quickly just by querying the Index. A low number of minimizer matches implies a fast chaining process which results in minimap2's fast runtime.

Hifimap on the other hand first computes Hits scores, applies some strategy to select regions and uses a 2-pointer merge-style algorithm to find hits between the query sequence and selected regions. This is simply redundant when dealing with sequences with a high percentage of singleton minimizers as the resulting list of minimizer matches that gets chained is about the same size as minimap2's. The main idea behind hifimap and the reason for the added overhead is to reduce this list down to a maximal constant size, independent of reference sequence length.

Mapping to short sequences can further be improved by precomputing minimap2-style indexes for each existing region and finding matches by just iterating query minimizers and querying indexes of corresponding regions, as discussed in section 1.3.4. This would deplete the need for the 2-pointer algorithm for finding matches used in the hifimap implementation.

Table 3.3 Synthetic target, length 10^9, **53.5%** singleton minimizers

	minimap2	hifimap	hifimap --save-mem	hifimap --approx
Indexing time	75.87s	190.7s	185.9s	167.1s
Mapping time	**346.7s**	**258.6s**	**313.2s**	**229.2s**
Peak RSS	11.66GB	23.81GB	14.21GB	14.21GB
# mappings	100000	100000	100000	100000
coverage t = 30bp	-	99.99%	99.99%	14.72%
coverage t = 100bp	-	100%	100%	44.96%
coverage t = 200bp	-	100%	100%	73.61%
coverage t = 400bp	-	100%	100%	96.14%

Table 3.4 Synthetic target, length $2 \cdot 10^9$, **33.11%** singleton minimizers

	minimap2	hifimap	hifimap --save-mem	hifimap --approx
Indexing time	136.4s	438.5s	407.1s	410.2s
Mapping time	**371.2s**	**241.8s**	**300.5s**	**199.4s**
Peak RSS	22.74GB	40.51GB	29.67GB	29.67GB
# mappings	100000	100000	100000	100000
coverage t = 30bp	-	99.96%	99.96%	9.13%
coverage t = 100bp	-	99.99%	99.99%	29.32%
coverage t = 200bp	-	100%	100%	52.52%
coverage t = 500bp	-	100%	100%	91.23%

A low percentage of singleton minimizers combined with large sequence length indicates highly repeating minimizers. As the number of repeating minimizers increases, the size of minimap2's list of minimizer matches that needs to be

chained increases. In contrast, the same list of matches has a constant upper bound in hifimap, hence we can observe positive results.

Hifimap should outperform minimap2 when the reference sequence has highly repeating minimizers. We benefit from the fact that minimizer repetition is mostly correlated with sequence length, so we expect and observe great performance for very large reference sequences which are of practical interest to us. Note that the largest known genome, *Paris japonica*, contains about 150 billion base pairs, 50 times more than the human genome.

Our approximation method didn't yield too satisfactory results, as its runtime is only a constant time away from the classic hifimap algorithm runtime at the price of accuracy that might be too high to be used in practice. It appears that as target sequence length increases, the approximation accuracy decreases.

Table 3.5 Target sequence: *E. coli*, length $4.6 \cdot 10^6$, 98.10% singleton minimizers
Number of HiFi reads: 95514

	minimap2	hifimap	hifimap --save-mem	hifimap --approx
Indexing time	0.316s	0.611s	0.557s	0.561s
Mapping time	**270.4s**	**169.2s**	**211.72s**	**128.99s**
Peak RSS	1.02GB	0.3GB	0.28GB	0.28GB
# mappings	98796	101526	101526	95514
coverage t = 30bp	-	96.21%	96.21%	27.31%
coverage t = 100bp	-	96.28%	96.28%	62.21%
coverage t = 200bp	-	96.36%	96.36%	73.96%
coverage t = 500bp	-	96.47%	96.47%	82.27%

We have to say we are pleasantly surprised with these results. It seems to go against our intuition that a high percentage of singletons benefits minimap2. Since this is a real dataset, it appears that hifimap adapts quite well to different read lengths. Note that this test was conducted on a different machine than all previous tests.

Conclusion

Algorithms need to keep up and adapt to the latest advancements in sequencing technologies in order to improve bioinformatics data pipelines. HiFi reads have an astounding accuracy greater than 99% and a narrow length distribution with the maximal mean length of 25k base pairs. We developed a novel mapping algorithm, hifimap, which takes advantage of these properties.

Hifimap partitions the reference sequence into logical overlapping regions, identifies the most prominent regions by computing tight upper bounds of chain lengths for each region and finds exact minimizer matches only on most prominent regions which are then chained into mapping results. The idea is to reduce the length on which finding matches and chaining is performed down to a maximal constant length in order to improve runtime. Partitioning the reference sequence into regions allowed us to employ the TFIDF method in order to filter noisy minimizers and also to approximate mapping positions even without the need for finding and chaining exact matches.

We have laid the theoretical foundation of the hifimap algorithm, discussed various algorithm details and presented our implementation. Results show that hifimap performs well on both small and large reference sequences while especially excelling on very large sequences with highly repeating minimizers.

References

[1] Heng Li, Minimap2: pairwise alignment for nucleotide sequences, *Bioinformatics*, Volume 34, Issue 18, 15 September 2018, Pages 3094–3100, https://doi.org/10.1093/bioinformatics/bty191

[2] Heng Li, Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences, *Bioinformatics*, Volume 32, Issue 14, 15 July 2016, Pages 2103–2110, https://doi.org/10.1093/bioinformatics/btw152

[3] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, James A. Yorke, Reducing storage requirements for biological sequence comparison, *Bioinformatics*, Volume 20, Issue 18, 12 December 2004, Pages 3363–3369, https://doi.org/10.1093/bioinformatics/bth408

[4] Robert Vaser, ram (2021, April) https://github.com/lbcb-sci/ram

Summary

Title: Rapid Alignment of High-Fidelity Sequencing Data

Keywords: Mapping, Alignment, HiFi reads, PacBio, hifimap, minimap2

Summary: Mapping DNA reads to a reference sequence is a common problem in bioinformatics and can be quite hard since sequences are often very large. High Fidelity reads, being the latest advancement in the field, have an astounding accuracy greater than 99% and a narrow length distribution with the maximal mean length of 25k base pairs. In this paper we propose a novel mapping algorithm optimized for HiFi reads, hifimap. We lay the theoretical foundations of the hifimap algorithm, discuss various algorithm details and present our implementation. Results show that hifimap performs well on both small and large reference sequences and in many cases outperforms minimap2 which is considered the industry standard.

Sažetak

Naslov: Brzo poravnanje visoko pouzdanih dugačkih očitanja

Ključne riječi: Mapiranje, Poravnanje, Hifi očitanja, PacBio, hifimap, minimap2

Sažetak: Poravnanje DNA očitanja na referentni genom jedan je od čestih problema u bioinformatici i može biti vrlo težak jer genomi mogu biti vrlo veliki. High Fidelity očitanja imaju zapanjujuću točnost iznad 99% i usku distribuciju duljina sa maksimalnom srednjom vrijednost od oko 25kbp. U ovom radu predstavljamo novi algoritam poravnanja optimiziran za HiFi očitanja, hifimap. Započinjemo sa postavljanjem teoretske podloge algoritma hifimap, zatim razmatramo razne algoritamske detalje i na kraju prezentiramo našu implementaciju. Rezultati su pokazali da hifimap ostvaruje dobre rezultate za male i velike referentne genome, a često i nadmašuje minimap2 koji je smatran industrijskim standardom.