Freie Universität Berlin
Department of Mathematics and Computer Science

Master's Thesis

Combining the PEX Algorithm with FM-Index Optimum Search Schemes for Read Mapping

Felix Leander Droop

06.05.2025

Supervisors
Prof. Dr. Knut Reinert   Simon Gene Gottlieb   Dr. René Rahn

Examinors
Prof. Dr. Knut Reinert   Prof. Dr. Wolfgang Mulzer

Disclaimers

This thesis was created using the Typst typesetting software [1].

The AI-based software tool DeepL Write [2] was used to improve stylistic aspects of the writing in this thesis.

Contents

Abstract

In this work, the PEX approximate string matching algorithm is modified to use longer seeds than its original version. In order to increase the seed length while maintaining the exactness property of the algorithm, the seeds are searched with a number of allowed errors. To this end, a novel bottom-up approach to constructing PEX trees is introduced and its correctness is proven. A prototype implementation that applying this new method to the long-read mapping problem is provided and analyzed. It uses optimum search schemes and an efficient bidirectional FM-Index implementation to improve the running time performance. The prototype is evaluated and compared to the original PEX algorithm as well as a state-of-the-art long-read mapper. The proposed method is faster and produces better results than the original PEX algorithm on biological data. However, it is not yet competitive with state-of-the-art solutions in terms of running time and memory consumption, and lacks the ability to detect structural variants, which is important for accurately mapping long-reads.

1 Introduction

1.1 DNA Sequencing

DNA sequencing is the process of determining the DNA base pair sequences in a biological sample. It is an indispensable method for most branches of microbiology and has been subject to significant technological advancements in recent decades. The first whole sequences of the human genome were published in 2001 as a result of years-long efforts that cost hundreds of millions of dollars [3], [4]. Nowadays, the typical cost of sequencing a human genome has fallen below 1000 dollars [5], as can be observed in the left part of Figure 1. Furthermore, technological advancements have made it possible to sequence a complete human genome in the matter of hours [6].

A wide range of biochemical protocols have been developed that offer new capabilities for a wide variety of research fields. Some of the most important examples are RNA-Seq for transcriptome analysis [8], Chip-Seq for DNA-protein interactions [9] and Hi-C for DNA-DNA interactions [10]. The importance of these methods to modern life sciences as a whole cannot be overstated and they ultimately rely on fast and efficient DNA sequencing technology.

However, the continuous improvements in cost-efficiency, throughput and application diversity of sequencing machines have led to an exponential increase in the amount of sequencing data available, which in turn has put enormous pressure on the software infrastructure that enables downstream data analysis. For example, from 1982 to the present, the number of bases in GenBank, an annotated collection of all publicly available

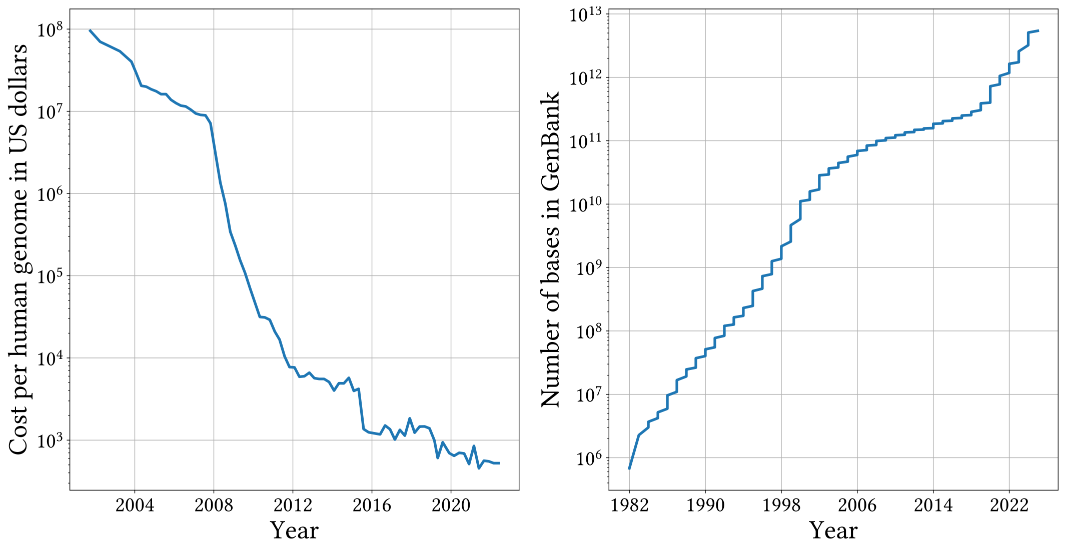

Figure 1: The number of bases stored in one of the largest data bases for biological sequence, GenBank, over the last 40 years [7] and the cost of sequencing a human genome over the last 20 years [5]. In both of the charts, the y-axis has a logarithmic scale. It can be observed that the amount of available data has increased and the cost of sequencing has decreased in exponential terms.

DNA sequences, has doubled approximately every 18 months [7]. Figure 1 shows the evolution of the number of bases in GenBank over the last 40 years.

Some of the most relevant algorithmic challenges in the processing of sequencing data are the fundamental steps of de-novo genome assembly and read mapping. Both of these methods deal with the fact that the long DNA molecules have to be fragmented into many pieces before they can be processed by sequencing machines. Raw sequencing data therefore consists of relatively short snippets of DNA sequence, so-called *reads*, rather than whole, continuous sequences. The latter have to be reconstructed either by *de-novo genome assembly*, which uses only the reads, or by *read mapping*, which additionally uses an existing, previously assembled reference genome. Nowadays, accurate reference genomes exist for most organisms. Therefore, read mapping is used in the majority of applications.

Accurate reconstruction of the whole, continuous sequences from the sample is only possible because the DNA is copied multiple times before being randomly fragmented. The random fragmentation makes it likely that the resulting reads at the same position of different DNA copies overlap with each other. This copying, also called *DNA amplification*, results in a large amount of data being generated. A whole-genome sequencing experiment for the human genome can generate around 100 gigabytes of raw sequencing data [11]. De-novo genome assembly and read mapping software has to be designed and optimized to be capable of handling data at this scale.

The DNA amplification and random fragmentation to create overlaps is particularly important for de-novo genome assembly, as no other information is available to aid in reconstructing the sequences of the biological sample. However, even with a guiding reference genome at hand, read mapping remains a challenging task. Reads can be approximately searched in the reference, but the presence of repetitive sequence and structural variants[1] in the genomes of most organisms, and errors introduced by sequencing machines can make it impossible to determine accurate and unique mapping locations (see Section 1.4 for a more detailed introduction into the read mapping problem). If such a location cannot be identified for a read, it is referred to as *unmapped*. Otherwise it is referred to as *mapped*.

1.2 Long-Read Sequencing

In the last 15 years, sequencing technologies have emerged that are capable of producing much longer reads than previously possible [12]. While established short-read sequencing systems supported read lengths of a few hundred base pairs (bps), the latest long-read sequencing platforms are capable of generating reads of up to a million bps in length [13]. Two different biochemical approaches have emerged as some of the most widely known technologies in this field. The first is nanopore sequencing [14], which works by passing a piece of DNA through a molecular pore and measuring the ion current around it, and the second one is single-molecule real-time sequencing [15], which is based on monitoring the activity of a polymerase enzyme as it synthesizes a strand of DNA.

[1]Structural variants are genomic variations that affect a few hundred up to millions of bps and are often the result of unequal DNA recombination. They include deletions, insertions, duplications, inversions and translocations of sequence.

The longer reads generated by these technologies often completely cover repetitive regions and structural variants. They therefore are instrumental in improving the resolution of genome assemblies and the quality of read mapping [13]. However, this advantage comes at the cost of a higher rate of sequencing errors. The most widely used long-read sequencers generate reads with error rates of around 8% [16], while typical short-read platforms are able to maintain error rates below 1% for their sequenced reads [17]. This issue can in part be mitigated by applying error correction methods [18], but it nevertheless poses a challenge for de-novo assembly and read mapping software.

1.3 Goal and Outline of this Work

The goal of this work is to develop and analyze a prototype of a long-read mapper. While most state-of-the-art long-read mappers make extensive use of heuristic algorithmic components (see Section 1.5), this work attempts to explore a different strategy and employ exact subroutines wherever possible. A worse running time performance is to be expected, but more accurate and under certain circumstances exact results are the possible advantages of this approach. The need for such work is demonstrated by the fact that even `minimap2` [19], [20], one of the most widely used read mapping tools, leaves about 10% of reads unmapped when handling nanopore sequencing data with high error rates (see Section 4.9). Figure 2 shows the input and output of read mappers, and contrasts the approach of this work to existing solutions.

This thesis is structured as follows. The remaining sections of this chapter provide an introduction to the read mapping problem and related terminology, as well as give an

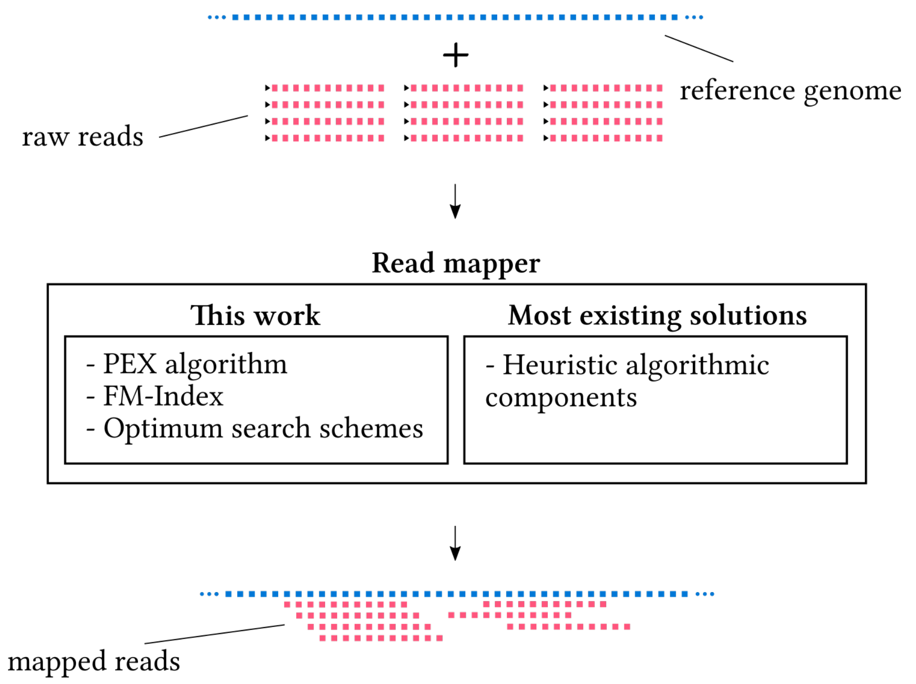

Figure 2: Read mappers take a reference genome and raw reads as input and find occurrences of the reads in the reference genome, i.e. *map* the reads. While most state-of-the-art read mappers use a wide range of heuristic algorithmic components, this work attempts to apply a number of exact methods to potentially achieve more accurate results.

overview of existing read mapping software. Next, the Preliminaries chapter introduces the existing algorithms used in the prototype read mapper of this work. It covers the PEX approximate pattern matching algorithm, the bidirectional FM-Index with search schemes, and the interval tree data structure.

The Methods chapter then describes how these algorithmic components are combined into a novel approach to long-read mapping. Additionally, it lists a number of smaller algorithmic ideas used to further improve this new method. The Results chapter then presents and analyzes benchmarks of the prototype read mapper implementation and compares it to a state-of-the-art read mapper. Finally, the Discussion chapter provides a summary and an evaluation of this work.

1.4 Read Mapping

This paragraph will give an overview of the basic terminology, challenges and algorithmic approaches involved in read mapping (sometimes also referred to as *read aligning*).

Given a collection of reads and a reference genome, the goal of the *biological* read mapping problem is to identify the position in the reference genome where each of the reads originated, assuming the reads were generated from a biological sample of the same species as the reference genome. In practice, it is not possible to solve this problem exactly, because of errors introduced by the sequencing machines and ambiguities due to repetitive sequence in the reference genomes.

Instead, existing tools attempt to solve or approximate the *mathematical* read mapping problem [21]. Here, the goal is to find sequence intervals in the reference genome that closely match the read sequence. These matching intervals, also called *matches*, are scored by a distance function that models the genetic variation between the reads and the reference genome, and how the reads may have been altered by sequencing errors.

A commonly used distance function is the *edit distance*. It counts the number of character insertions, deletions and substitutions that are required to transform one sequence into the other. Other distance functions exist that attempt to model the biological relationship of the sequences more closely. Linear scoring functions assign constant weights to the costs of the edit distance operations. Statistically less likely operations can be modelled as having a higher cost. Linear-affine cost functions introduce a gap-opening penalty that only affects the first operation of successive insertions or deletions (*indels*). In this model, many short indels amount to a higher cost than fewer longer ones, the latter being more likely to occur in the evolution of biological sequences. Finally, when dealing with long-read data, it may be biologically relevant to include the possibility of structural variants in the modeling process. The resulting scoring functions are often empirically derived and complex [20].

The distance functions are not the only way in which read mapping software takes into account the unique characteristics of biological sequences. Genomes often contain vast regions of highly repetitive sequence that can pose challenges to or even completely break algorithms that rely on certain statistical properties of the sequences. All read mappers apply strategies to deal with these so-called *repeats* or rely on the preprocessing by other software [22]. Furthermore, the reference genome represents only one of the two strands of the double-stranded DNA molecule. However, reads are generated from

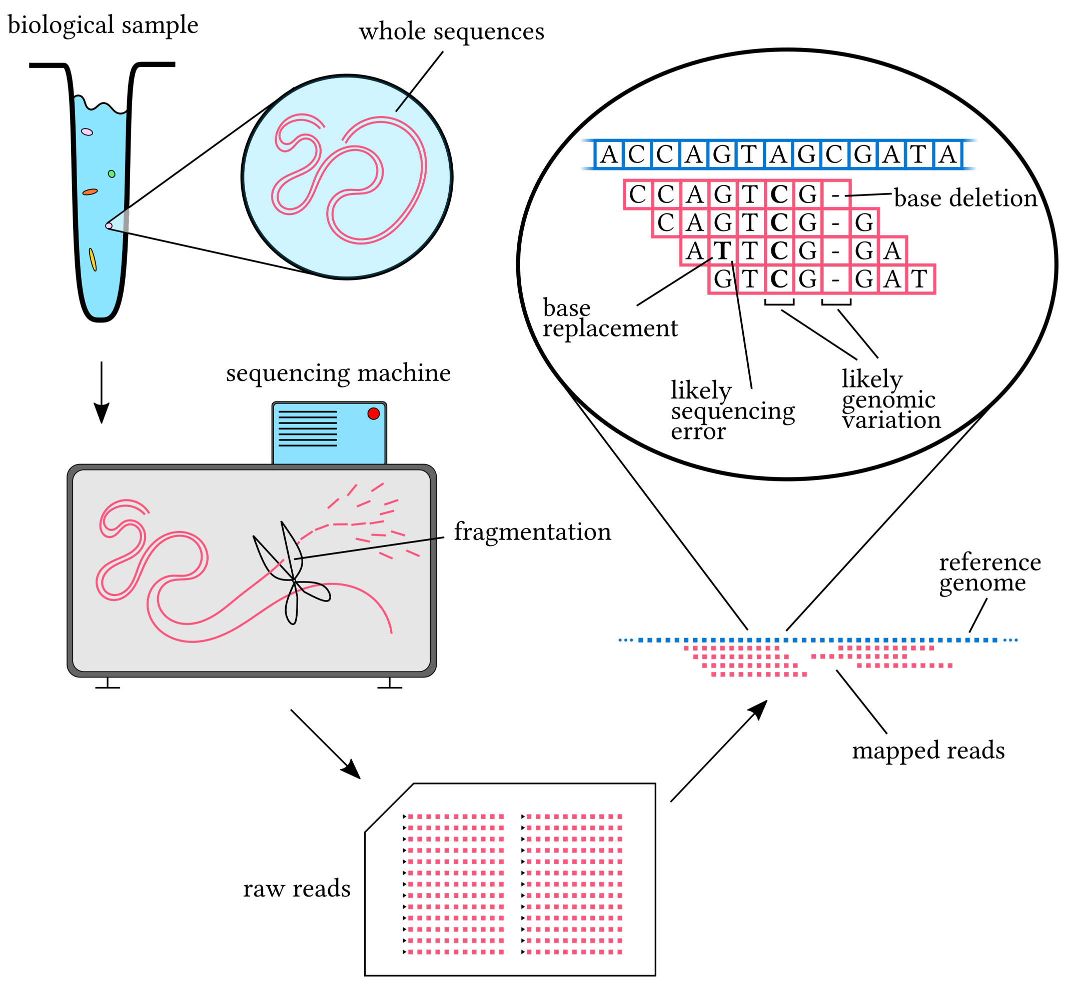

Figure 3: An overview of a typical read mapping workflow. Whole DNA sequences are extracted from a biological sample and fragmented by a sequencing machine. The raw reads are aligned to a previously assembled reference genome. If only a small number of reads in an alignment column differ from the reference, a sequencing error in these reads is assumed. However, if most of the reads in an alignment column differ, biological variation between the sample and the reference genome is likely to have been found.

both strands by the sequencing machines. Reads that originate from the strand not represented by the reference genome therefore have to be transformed into their counterpart sequence on the other strand. Each one of the four DNA bases has a specific complement base, which it is bound to on the opposite strand. In addition, the molecular backbones of the two strands are oriented in opposite directions and reads are always generated following a specific one of the two molecular orientations. Therefore, the counterpart of a read on the opposite strand can be obtained by reversing its sequence and replacing all bases with their complement. This transformation is called the *reverse complement*. Read mapping tools have to be aware of this phenomenon, otherwise about half of all reads could not be mapped correctly.

Depending on the downstream application, different sets of matches are desired as the result of read mapping. The prototype read mapper implemented in this work attempts to find all matches with edit distance scores below a given threshold. Another common goal is to identify a single or a few matching regions with optimal or near-optimal distance to the read.

The output of read mappers usually includes not only the locations and scores of the matches, but also contains a representation of how the read sequence has to be arranged next to the reference genome at the matching location to achieve the given score. This representation is called an *alignment*. Read alignments are necessary for many types of sequencing data analysis, because they allow sequencing errors to be distinguished from biologically relevant sequence variation. This works by comparing the alignments of multiple reads at the same reference genome position, as different reads are unlikely to contain the exact same sequencing errors. An example of this mechanism can be seen in Figure 3. If the detailed alignment information is not needed, read mappers may choose not to report it to save computational resources.

1.4.1 Common Algorithmic Design Frameworks for Read Mapping

From an algorithmic point of view, the design of read mappers is heavily influenced by the large size of most reference genomes, which precludes the use of data structures with superlinear space requirements, and the high number of reads, which incentivizes preprocessing steps such as building text indices on the reference genome to reduce the running time required per read. The most common framework used to overcome these limitations is called *seed-extend* [13]. It consists of the following basic steps. First, the reads are transformed into a set of elements, called seeds. The most basic form of seeds are simply continuous subsequences of the read. An abundance of more complex seeding strategies exists, some of which will be touched upon in Section 1.5. In the next step, a previously built index data structure is used to locate occurrences of these seeds in the reference genome. These occurrences are called *anchors*. Finally, an often computationally expensive algorithm is run on a small window around the anchor to examine whether it can be extended to an occurrence of the whole read. This last step is often referred to as the *seed extension*, *verification* of a full read occurrence, or simply called the *alignment*.

The handling of repetitive sequence in the reference genome can be done at different steps of this framework. Seeds that occur in repetitive regions can be excluded already while constructing the index, during the index-lookup or in an additional filtering step. Most of the existing strategies rely on the assumption that downstream applications are not interested in read alignments within repetitive regions, which is often warranted. Therefore, not aligning reads to repetitive regions can be justified.

In recent years, the increasing relevance of long-read sequencing has led to a change in the *seed-extend* paradigm. New methods have started to incorporate another so-called *chaining* step [13]. Here, all anchors produced by a single read are examined together and merged into so-called chains if they are likely to belong to the same read occurrence. Verifications are then run once per chain instead of verifying read occurrences for every single anchor. The additional step can save significant amounts of computational resources, as running one verification per anchor can be costly when mapping long-reads. This is due to the large number of seeds produced by long-reads and dramatically more expensive verification computations on longer sequences. The new framework is called *seed-chain-extend*. A visualization of its core steps can be observed in Figure 4.

1.5 Existing Read Mapping Software

Over the years, many read mapping tools have been developed, continuously refining their algorithmic approaches and heuristics. It would be impossible to mention them all in this work. Instead, this section will list the most influential software and briefly explain its applied techniques, with a focus on long-read mapping. The selection of tools to be included is based on the review papers [13] and [23].

The most widely used read mappers for traditional short-read sequencing data are `bowtie2` [24] and `bwa-mem` [25]. Both use an FM-index (see Section 2.2) to search continuous subsequences of the read. While `bowtie2` is an implementation of the original seed-extend framework, `bwa-mem` already includes a simple, greedy chaining step.

`GraphMap` [26] was one of the first dedicated long-read mappers. It uses *spaced seeds* that are stored in a hash-based index. Spaced seeds are short subsequences in which a given set of positions are allowed to differ between read and reference. These positions are called *wildcards* and can make the seeding procedure more robust against small numbers of edit distance errors. For each position in the reference, one seed is stored in the index and for each position in the read, three seeds are looked up to account for the different edit distance operations using different wildcard patterns. `GraphMap` then employs an incremental funnel-like approach in which the anchors anchors are chained and candidate chains are continuously refined and excluded using graph-based algorithms.

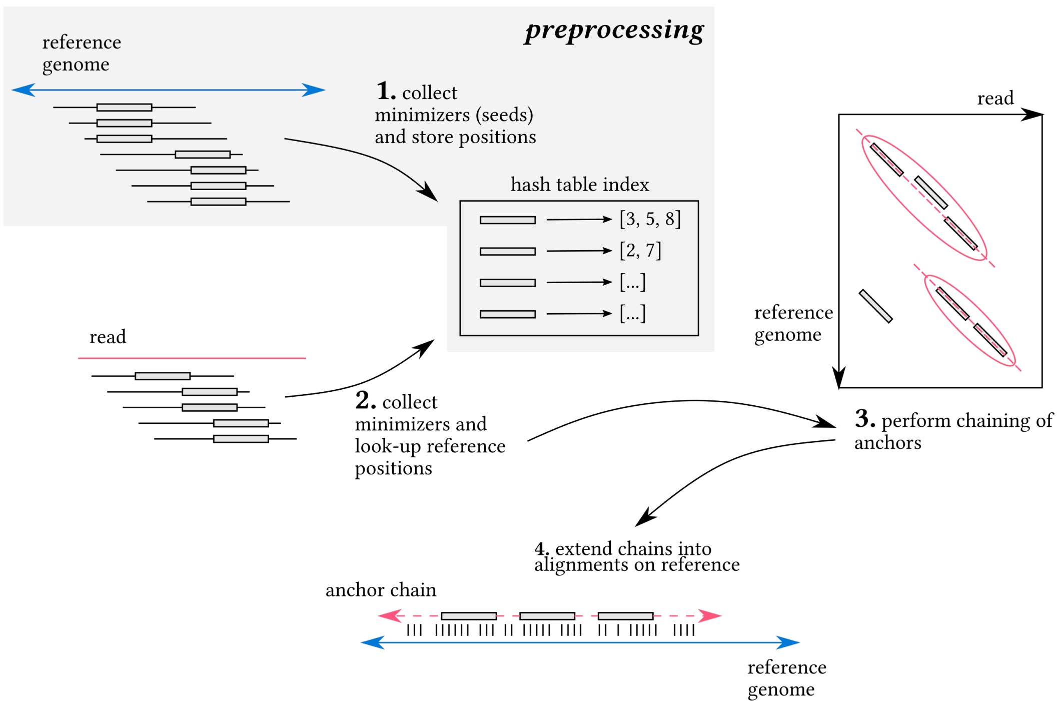
preprocessing
reference genome
1. collect minimizers (seeds) and store positions
hash table index
[3, 5, 8]
[2, 7]
[...]
[...]
read
2. collect minimizers and look-up reference positions
read
reference genome
3. perform chaining of anchors
4. extend chains into alignments on reference
anchor chain
reference genome

Figure 4: The structure of a seed-chain-extend read mapper using the example of `minimap2`. In the first preprocessing step, the minimizer seeds are collected from the reference genome and their positions are stored in the hash table index. Then, for every read, anchors are obtained by looking up the reference positions of its minimizers in the index. In the third step, anchors are joined into chains. Finally, the chains are extended into full read alignments.

NGMLR [27] is a long-read mapper with a focus on accurate resolution of structural variants. It splits the long-reads into 256 bp long subsequences and aligns them independently. These large anchors are then used as input to a chaining algorithm that can detect structural variant patterns.

minimap2 [19], [20] is a general purpose read mapping tool that works on both short and long-read sequencing data. It is one of the most widely used read mappers and the de facto standard for long-read mapping. As of January 27th, 2025, it has more than seven times as many citations (10367) as the second most cited long-read mapper in this list, NGMLR (1438), which was published together with another widely used tool for structural variant identification. Because of this high relevance, minimap2 is chosen as the gold standard for the evaluation in this work. In the following paragraphs, it will be described in greater detail than the other tools.

minimap2 follows a typical seed-chain-extend approach, which can be observed in Figure 4. For seeding, it uses *minimizers* [28], which are a technique to systematically sample q-grams[2] from sequences. Given $w \geq q$, then for every w-sized window of the sequence, the lexicographically smallest q-gram out of all q-grams inside the window is chosen as a representative. When applied to DNA or RNA sequences, the reverse complements of the q-grams are additionally considered for the minimum. The window is shifted by one in each step, i.e. the windows overlap.

To build the index data structure of minimap2, the minimizers are computed for the whole reference genome and stored in a hash table. The key is the minimizer hash and the value is a list of positions where it occurs. Using minimizers has several advantages. It reduces the space requirements of the index compared to using all q-grams of the reference genome, because the same representative is often chosen for adjacent windows. Secondly, it has a good robustness against errors as demonstrated in [28]. By considering reverse complement q-grams, it is independent of DNA strands, which further reduces the space requirements. Finally, locating seeds on the reference can be done very quickly, since only a single hash computation and memory look-up is needed to obtain all occurrences. As a side note, recent versions of GraphMap have also shifted to using a minimizer-based index [29].

For chaining, minimap2 uses a dynamic programming algorithm, which is accelerated by simple heuristics. It is aware of structural variants and can map different parts of the long-reads to distant regions of the reference genome. This capability is referred to as *split-read* or *chimeric* alignment. The algorithm that computes base-level alignments uses an affine-linear cost function and employs a variety of heuristics. All of minimap2's algorithms are well optimized, making it the fastest widely used read mapper to date [23].

In the last years, several read mappers have been developed that are based on minimap2 and attempt to improve its seeding strategy. Two notable ones are winnowmap [30] and BLEND [31]. winnowmap introduced the weighted minimizer technique. It assigns a weight to q-grams based on how frequently they occur in the reference. Highly frequent q-grams are likely to be contained in repetitive sequence and are given weight that reduces their probability of being chosen as the representative in the minimizer calculation. This

[2]A q-gram is a continuous sequence of length q. q-grams are often overlapping subsequences of longer sequences and used to transform a sequence into a set while preserving some of the sequential information.

strategy improved the alignment accuracy and index compression. It has been integrated into recent versions of minimap2 [20]. BLEND is based on a more complex so-called *fuzzy seeding* approach. It can significantly improve the running time performance and the memory footprint of minimap2, but focuses on very novel long-read sequencing platforms with low error rates. This makes makes it different in scope compared to this work, which explores a method targeted at more traditional long-read data sets with high error rates.

lra [32] is a dedicated long-read mapper that attempts to compete with minimap2 by incorporating additional techniques into the seed-chain-extend framework. Firstly, it uses a version of weighted minimizer sampling that is applied locally to partitions of the reference sequence. Secondly, it uses a sophisticated chaining algorithm that is aware of structural variants. To speed up this computationally expensive algorithm, the anchors are pre-clustered using a simple heuristic. The result is a slightly slower and more memory consuming method than minimap2, with results of comparable quality. However, the two read mappers differ in the exact set of alignments they detect. This opens up the possibility of using both tools and obtaining a consensus result. An independent benchmark [23] of long-read mappers recently came to the same conclusion and recommended lra and NLGMLR as complementary tools to minimap2 or winnowmap.

2 Preliminaries

This section provides detailed explanations of the existing algorithms and data structures that form the basis of this work.

2.1 The PEX Algorithm

The PEX algorithm [33] is a pattern matching algorithm that finds all occurrences of a pattern in a text with an edit distance of at most k, i.e. k errors. It is an *approximate* string matching algorithm in the sense that it allows for errors in the pattern, but is *exact* in its guarantee to find every single occurrence of the pattern within the edit distance bound.

PEX is based on two main ideas, *pigeonhole seeding* and *hierarchical verification*. The former works by partitioning the pattern into $k + 1$ pieces, which function as seeds, and searching each one of them with 0 allowed errors in the text. This can be done with exact multi-pattern search algorithms. If there is an occurrence with at most k errors in the text, at least one of the pieces has to match exactly within that occurrence, due to the pigeonhole principle.

One could then run an algorithm to verify the existence of an occurrence of the full pattern in a sufficiently large window around each piece that was found in the text. However, this may be practically inefficient for longer patterns, since the verification algorithms for edit distance generally have at least quadratic running times. In the case of *false positive anchors*, a lot of unnecessary computation would be done. In this work, false positive anchors refer to anchors which are not located within a complete pattern occurrence. The *hierarchical verification* strategy is a way to improve on this inefficiency.

It works by iteratively increasing the size of the reference window in which an occurrence of an increasingly large part of the pattern is verified. Care must be taken not to break the pigeonhole logic when doing this. To this end, a generalized version of the pigeonhole principle (Lemma 1) is applied to construct a tree that defines how the size of the verification window is iteratively increased and how many errors are allowed for the respective verification run. This tree will be referred to as the *PEX tree* throughout this work.

Lemma 1 *(Generalized pigeonhole lemma, applied to approximate string matching)* Assume there exists an occurrence *occ* in the text of a pattern p with k errors (including indels). Let $P = P_1, \ldots, P_l$ be a partition of the pattern into subpieces and $a_1, \ldots, a_l$ be non-negative integers such that $A = \sum_{i=1}^{l} a_i$. Then, for at least one $i \in \{1, \ldots, l\}$, the occurrence of p in the text must contain a substring that matches P_i with at most $\left\lfloor \frac{a_i \cdot k}{A} \right\rfloor$ errors.

Proof First, let r_i denote the amount by which the expression $\frac{a_i \cdot k}{A}$ is reduced when applying the floor function:

$$r_i = \frac{a_i \cdot k}{A} - \left\lfloor \frac{a_i \cdot k}{A} \right\rfloor < 1$$

Then, assume that for every $i \in \{1, ..., l\}$, p_i matches any substring of *occ* with at least $\left\lfloor \frac{a_i \cdot k}{A} \right\rfloor + 1$ errors. For the total number of errors of *occ*, the following lower bound is acquired:

$$\begin{aligned}
&\sum_{i=1}^{l} 1 + \left\lfloor \frac{a_i \cdot k}{A} \right\rfloor \\
&= \sum_{i=1}^{l} 1 + \frac{a_i \cdot k}{A} - r_i \\
&= l + \frac{k}{A} \cdot \sum_{i=1}^{l} a_i - \sum_{i=1}^{l} r_i \\
&= l + k - \sum_{i=1}^{l} r_i \\
&> l + k - l \\
&= k
\end{aligned}$$

This is a contradiction, because the total number of errors of *occ* is exactly k and therefore cannot be greater than k. □

Each node of the PEX tree is associated with a sequence range of the pattern and a number of allowed errors. The generalized pigeonhole lemma is used to derive the formula for setting the number of errors in the PEX tree construction algorithm (see Algorithm 1, lines 10-11). It works by recursively splitting the pattern range in half and setting the number of allowed errors accordingly (lines 7-13). When the number of errors of a node reaches 0, the base case of the recursion is triggered and the node becomes a leaf (lines 5-6).

Algorithm 1: Recursive top-down PEX tree construction for pattern p and k errors.

```
1  ConstructTree ( $p = p[1]...p[m]$ , $k$ ) :
2      AddNodes ( $p$ , $k$ , $\perp$ , $\left\lfloor \frac{m}{e+1} \right\rfloor$ )
3  AddNodes ( $p = p[i]...p[j]$ , $k$ , parent_node , piece_length )
4      Add node curr_node to tree with parent parent_node, pattern range $[i, j]$ and $k$ errors
5      if $k == 0$ :
6          Mark curr_node as leaf
7      else :
8          let num_leaves_left_subtree = $\left\lceil \frac{k+1}{2} \right\rceil$
9          let split_pos = $i$ + num_leaves_left_subtree · piece_length
10         let k_left_child = $\left\lfloor \frac{num\_leaves\_left\_subtree \cdot k}{k+1} \right\rfloor$
11         let k_right_child = $\left\lfloor \frac{(k+1 - num\_leaves\_left\_subtree) \cdot k}{k+1} \right\rfloor$
12         AddNodes ( $p[i]...p[split\_pos - 1]$ , k_left_child , curr_node , piece_length )
13         AddNodes ( $p[split\_pos]...p[j]$ , k_right_child , curr_node , piece_length )
```

The search phase of the PEX algorithm works as follows (see Algorithm 2). First, the pattern ranges of all leaves of the PEX tree are searched using an exact multi-pattern search algorithm (line 3). Then, whenever the sequence of a leaf is found, the hierarchical verification procedure walks upwards through the tree and verifies the pattern range of each node it visits, on an appropriate window in the text with the number of errors stored in the node (lines 8-16). The text windows are determined by taking into account where the found leaf is located in the whole pattern and adding room for indel errors on both ends of the window (lines 10-11). The algorithm continues walking up the tree until a verification is unsuccessful or it reaches the root (line 8). In the latter case, a complete occurrence of the pattern has been found (lines 17-18). Figure 5 shows how this is done for an example pattern $p =$ aaabbbcccddd. The top half of the figure shows how an anchor is found in a complete occurrence of the pattern, which is then verified using the hierarchical verification. The bottom half shows how the hierarchical verification can stop before reaching the root of the tree in the case of a false positive anchor.

This procedure still guarantees that all occurrences of the pattern with at most k errors are found, because the formula for setting the errors of the PEX tree nodes was derived from the generalized pigeonhole principle. Thus, Lemma 1 can be applied recursively to each node and its children, starting from the root. In summary, PEX is a correct approximate pattern matching algorithm, which has a tolerance to false positive anchors and is based on the generalized pigeonhole principle.

Algorithm 2: The PEX approximate search algorithm for pattern p, text t and k errors

```
1  PEX ( $p = p[1]...p[m]$ , $t = t[1]...t[n]$ , $k$ ) :
2      let tree = ConstructTree ( $p$ , $k$ )
3      let anchors = output of multi-pattern search for pattern ranges of all leaves of tree

4      for (anchor_leaf, anchor_pos) in anchors :
5          let [leaf_start, _] = anchor_leaf.pattern_range
6          let curr_node = anchor_leaf.parent
7          let is_candidate = true

8          while is_candidate and curr_node != ⊥ :
9              let [curr_start, curr_end] = curr_node.pattern_range
10             let text_start = anchor_pos - (leaf_start - curr_start) - curr_node.errors
11             let text_end = anchor_pos + (curr_end - leaf_start + 1) + curr_node.errors

12             if occurrence of $p[curr\_start]...p[curr\_end]$ exists in
               $t[text\_start]...t[text\_end]$ with at most curr_node.errors errors:
13                 curr_node = curr_node.parent
14             else :
15                 is_candidate = false

16         if is_candidate :
17             Report position where all of $p$ was found
```

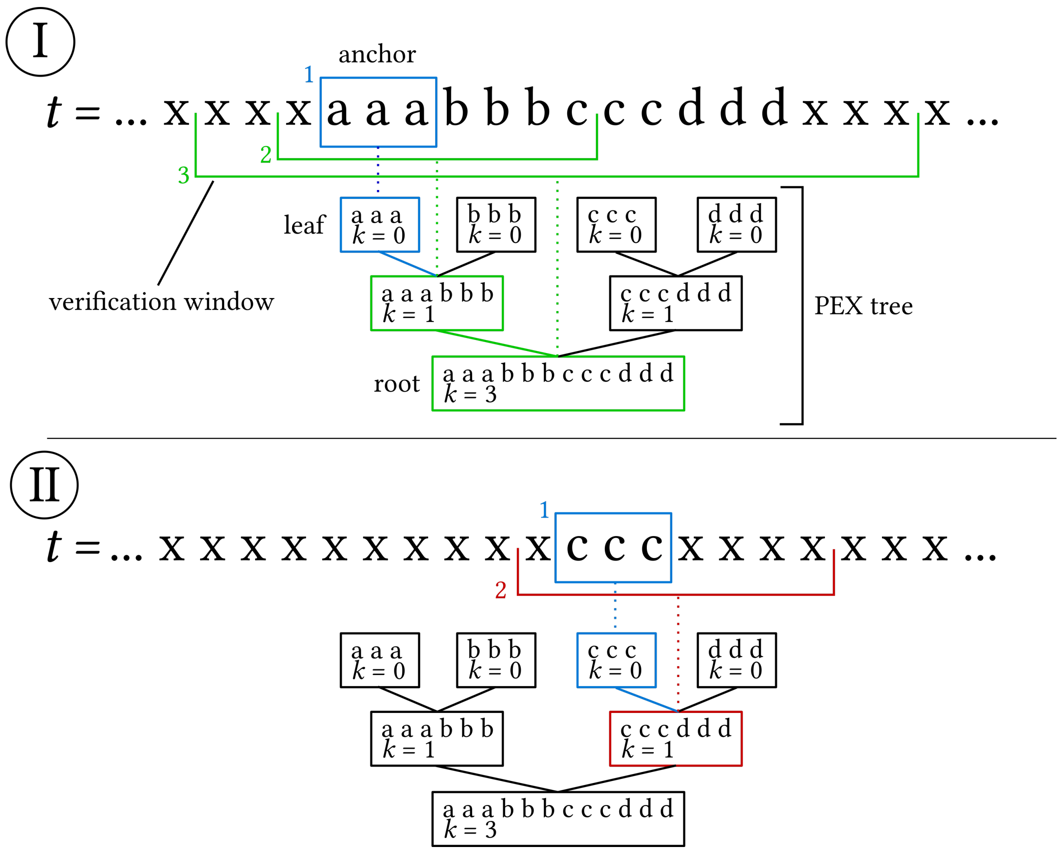

Figure 5: This figure shows how the PEX tree for the pattern $p = \text{aaabbbcccddd}$ and $k = 3$ guides the hierarchical verification in two different parts of the text t. In the top half, the seed "aaa" is found and the anchor (1) is extended to a full occurrence of p using the hierarchical verification along the green part of the tree. The corresponding verification windows are shown next to the sequence (2,3). On the bottom half, the seed "ccc" is found (1). The anchor cannot be extended to a full occurrence of p, because the verification of the red node of the tree fails (2). In this case, the algorithm did not need to run the verification algorithm for all of p.

2.2 FM-Index

The FM-Index [34] is a full-text index data structure. It is built on a text $t = t[1]\dots t[n]$ over an ordered alphabet $\Sigma, |\Sigma| = \sigma$. To simplify the following description, it is is assumed that t is terminated by a unique, smallest character called *sentinel*, whichis denoted by $t[n] = \$$. The FM-Index supports two types of operations. Given a pattern $p = p[1]\dots p[m]$ over $\Sigma \setminus \{\$\}$, the `count` operation returns the number of exact occurrences of p in t and has a running time in $O(m)$. The `locate` operation has to be preceded by a `count` operation and returns the positions in t of the occurrences of p. Its running time is in $O(K)$, where K is the value returned by the `count` operation. The space requirements of the FM-Index depend on the implementation and are discussed later in this section. In practice and for small alphabets such as the DNA alphabet with $\sigma = 5$, the data structure can be stored in as little as 3-4 times the size of t without a significant degradation of running time performance.

Internally, the FM-Index consists of the *suffix array*, the *count vector* and the *occurrence table* of the *Burrows-Wheeler-Transform* [35] of t. All of these components will be introduced in the following paragraphs.

The i*-th suffix* of t is the substring $t[i]...t[n]$. The suffix array of t is an array containing the numbers 1 to n, which are sorted according to the lexicographical order of the corresponding suffixes of t. For example, if the suffix array starts with i_1 and i_2, then the i_1-th suffix is the lexicographically smallest suffix out of all suffixes of t and the i_2-th suffix is the second smallest. It follows that the number of bits required to store the suffix array is $n \cdot \lceil log_2\, n \rceil$. A fundamental property of the suffix array is that all suffixes starting with a common series of characters are grouped together in a contiguous interval, due to the lexicographical ordering. Using two binary searches, it is possible to find the boundaries of this interval for any given pattern of characters. From the boundaries of the interval, the answer to the `count` operation can be trivially computed. The running time of this implementation of the operation is in $O(m \cdot log\, n)$. A scan through the interval delivers the result for `locate` in $O(K)$ running time. Using heuristics and an auxiliary data structure called the *lcp array*, it is possible to improve the running time of `count` to $O(m + log\, n)$. The exact details of how this is done are beyond the scope of this work.

The FM-Index uses the suffix array only to implement the `locate` operation and obtains the bounds of the interval containing the indices of all occurrences of the pattern without binary searches in $O(m)$ time. The central data structure that makes this possible is the Burrows-Wheeler-Transform (BWT) of t. It is based on a concept similar to the suffix array, the Burrows-Wheeler-Matrix (BWM). The rows of the BWM are the lexicographically sorted *rotations* of t. The i*-th rotation* of t is defined as the string $t[i]...t[n]\, t[1]...t[i-1]$ if $i > 1$ and t otherwise. From this definition it can be observed that each column of the BWM is a permutation of t. The BWT is the last column of the BWM and is also denoted as L in the context of the BWM. The first column of the BWM is also denoted as F. Given the suffix array *suf*, the BWT can alternatively be described as follows:

$$BWT_t[i] = \begin{cases} t\,[suf[i] - 1] & if\ suf[i] > 1 \\ \$ & \text{otherwise} \end{cases}$$

To implement the `count` operation, the FM-Index does not use the BWT directly. Instead, it stores two auxiliary data structures. For a character $c \in \Sigma$, the count vector $C[c]$ is defined as the number of characters in t that are lexicographically smaller than c. Secondly, for a character $c \in \Sigma$ and an index i, $1 \leq i \leq n$, the occurrence table $Occ[c, i]$ is defined as the number of occurrences of c in $BWT_t[1]...BWT_t[i-1]$ if $i > 1$ and 0 otherwise. The implementation of the `count` operation is based on these two data structures and uses the *rank preservation* property (Lemma 2). In general, the *rank* of an instance of the character c at a position i of a sequence S refers to the number of occurrences of c in the subsequence $S[1]...S[i]$.

Lemma 2 *(Rank preservation)* The occurrences of the i-th character of t have the same rank in the first and last columns of the BWM.
In other words, let $t[i] = c$ and $t[j] = c$ be two instances of the same non-sentinel character in t. If $t[i]$ occurs at an earlier position than $t[j]$ in L then $t[j]$ also occurs at an earlier position than $t[i]$ in F.

Proof The order of occurrence of $t[i]$ and $t[j]$ in F depends on the lexicographical order of the i-th and j-th rotations. However, since the two characters are equivalent and the text is terminated by the unique sentinel $t[n] = \$$, their relative order effectively depends only on the order of the substrings $t[i+1]...t[n]$ and $t[j+1]...t[n]$. There can be no tie, because both substrings have different lengths. Next, the order in which the two characters appear in L depends on the order of the same substrings. This is due to the way the rotations of t are defined. For example, if a row is terminated with $t[i]$, then it starts with the substring $t[i+1]...t[n]$. It follows that the relative order of $t[i]$ and $t[j]$ in the first and last columns of the BWM is determined by the order of the same substrings of t and therefore must be equivalent. A visualization of this argument can be found in Figure 6. □

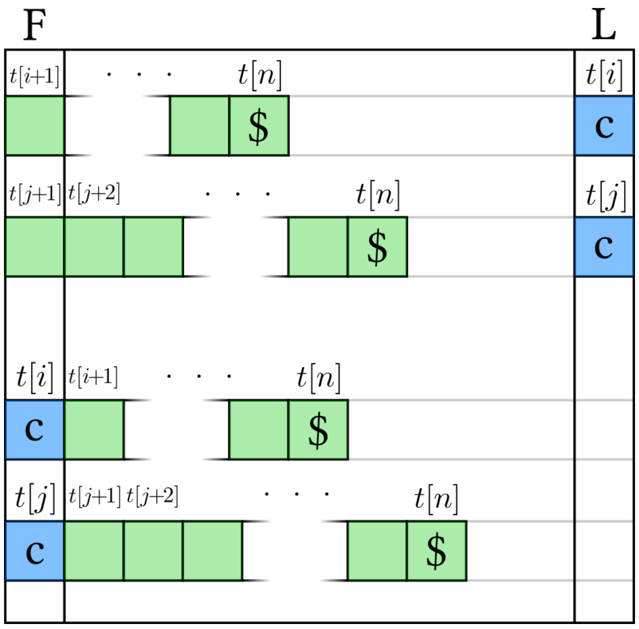

Figure 6: A visualization of the rank preservation property. A BWM with its first (F) and last column (L, the BWT) is shown. The occurrences of the characters $t[i] = c$ and $t[j] = c$ in the two columns are highlighted in blue. It can be observed that the relative order of the two characters in both columns depends on the lexicographic order of the same respective substrings of t, highlighted in green. In this visualization, $t[i+1]$ and $t[j+1]$ are lexicographically smaller or equal to c, which does not always have to be the case.

Using Lemma 2, the FM-Index can perform the *pattern extension* step, which forms the basis of the implementation of `count`. Given an interval of BWM row indices $[l, r)$ containing all rotations starting with a given pattern cs over Σ and a character $c_{next} \in \Sigma$, the interval $[l_{next}, r_{next})$ containing the rotations starting with the pattern $c_{next}\ cs$ can be obtained using the following calculations. The formula that is applied to l and r is called *LF-Mapping*:

$$l_{next} = C[c_{next}] + Occ\ [c_{next}, l]$$
$$r_{next} = C[c_{next}] + Occ\ [c_{next}, r]$$

This works because of the rank preservation and another property that follows directly from the definition of rotations of t. It states that the rotations contained in the interval $[l_{next}, r_{next})$ exactly correspond to those rotations in $[l, r)$ terminated by c_{next}. They are merely rotated by one to the right. Therefore all that needs to be done to derive $[l_{next}, r_{next})$ from $[l, r)$ is to identify the rows in $[l, r)$ with instances of c_{next} in the last column and locate these exact instances of c_{next} in the first column. This can be done efficiently using the rank preservation property. First, the start of the BWM row interval containing rotations starting with c_{next} has to be determined by looking up $C[c_{next}]$. Then, the value $Occ\ [c_{next}, l]$ is added to skip the occurrences of c_{next} before l in the last column and acquire l_{next}. To compute r_{next}, the occurrences of c_{next} in the last column within $[l, r)$ also have to be skipped by instead adding the value $Occ\ [c_{next}, r]$. It follows that the above formulae are a correct way of deriving $[l_{next}, r_{next})$ from $[l, r)$. A visualization of this procedure can be observed in Figure 7.

Using these formulae, the FM-Index can compute the BWM row interval containing all rotations starting with the pattern by beginning with the interval $[1, n+1)$ and the empty pattern, and incrementally adding the characters of p from the back to front. The final resulting BWM row interval is equivalent to the suffix array interval that is needed to extract the occurrences of p in the `locate` operation. This implementation of `count` is called *backwards search*. It has a running time in $O(m)$, because the lookups $C[c]$ and $Occ\ [c, i]$ can be implemented in $O(1)$ time, given enough space to store precomputed values for all possible inputs.

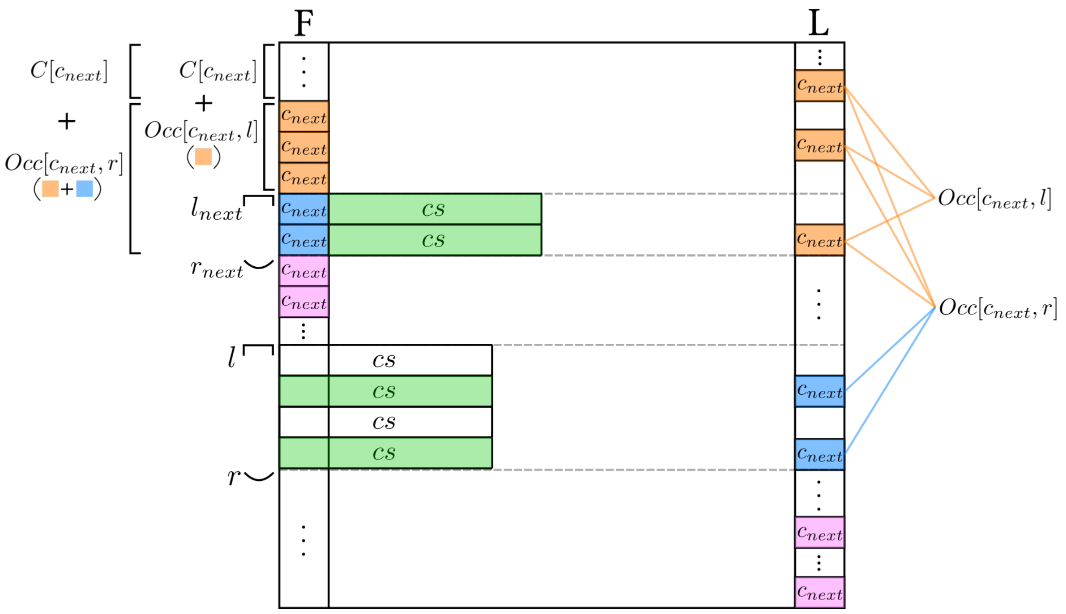

Figure 7: A visualization of the FM-Index pattern extension step. All occurrences of the character c_{next} in the first (F) and last column (L, the BWT) of a BWM are highlighted. The occurrences located above the interval $[l, r)$ in the last column and counted by the term $Occ\ [c_{next}, l]$ are colored orange. The occurrences of c_{next} within $[l, r)$ that account for the difference between $Occ\ [c_{next}, l]$ and $Occ\ [c_{next}, r]$ are shown in blue. The pink occurrences of c_{next} are located below the interval $[l, r)$ in L. Instances of the pattern *cs* that are preceded in t by c_{next} are highlighted in green. The upper left part of the image shows how the interval $[l_{next}, r_{next})$ is derived using the LF-Mapping formula.

For the count vector C, storing all possible values is not an issue, since only $\sigma \cdot \lceil log_2\, n \rceil$ bits are required. However, a naive implementation of the occurrence table *Occ* would require $\sigma \cdot n \cdot \lceil log_2\, n \rceil$ bits of memory, which would be impractical for most applications. Many ways of implementing the occurrence table have been developed, attempting to reduce its memory consumption while maintaining a fast query running time. The FM-Index used in this work applies one of the latest and most efficient such implementations, the *EPR-dictionaries* [36]. They are based on the support data structures of bit vectors with rank support[3], and bit-parallel operations that operate directly on the BWT. With a space requirement of $n \cdot \lceil log_2\, \sigma \rceil + o(n \cdot \sigma \cdot log\, \sigma)$ bits for small alphabets, they are competitive with, but not among, the most space-efficient known implementations of the occurrence table. However, they offer query times that are faster than previous methods with comparable space complexity. The exact details of how the EPR-dictionaries work are beyond the scope of this thesis.

Another issue for the space consumption of the FM-Index is storing the suffix array for implementing the `locate` operation. This problem can be alleviated by subsampling the suffix array. If a value that is not stored is needed, the LF-Mapping formula can be applied until a retained entry is reached. This strategy allows for a direct trade-off between space consumption and running time, and the subsampling ratio can be chosen according to the needs of the specific application.

2.2.1 Bidirectional FM-Index

There exist applications that require dynamic control over how the pattern is extended during the backwards search, beyond the `count` and `locate` interface. Such dynamic control is possible without any algorithmic changes and depends only on the implementation. However, the ability to extend the pattern in both directions may additionally be necessary or beneficial. For this purpose, the *bidirectional* FM-Index [37] has been developed. It allows the search to start at any position in p and supports extending the currently searched part of the pattern in both directions.

The basis of the bidirectional FM-Index is to store two occurrence tables, *Occ* for the BWT of t and Occ^{rev} for the BWT of the reversed t. Two intervals, $[l, r)$ for the BWM of t and $[l^{rev}, r^{rev})$ for the BWM of the reversed t, are maintained. In the following, the superscript *rev* will mark entities that have already been introduced, except that with the superscript, they refer to the data structures built on the reversed t. Due to the availability of the two occurrence tables and intervals, adding a character to the front or back of the currently searched part of the pattern is a normal pattern extension for one of the two intervals, using the corresponding occurrence table. However, keeping the other interval synchronized requires an additional data structure, *Prefix-Occ*. Given $c \in \Sigma$ and i with $1 \leq i \leq n$, $Prefix\text{-}Occ\ [c, i]$ is defined as the number of occurrences of lexicographically smaller characters than c in $BWT_t[1]...BWT_t[i-1]$ if $i > 1$ and 0 otherwise. $Prefix\text{-}Occ^{rev}$ can be used according to Lemma 3 to synchronize the interval $[l, r)$, which cannot use a normal pattern extension if a character is added to the front of the currently searched part of the pattern. If a character is added to the back, $Prefix\text{-}Occ$ is used analogously to synchronize $[l^{rev}, r^{rev})$.

[3]Given an index of the bit vector, the `rank` operation returns the number of 1's in the vector up to that index. The support data structures that allow these queries to be computed in constant time consist of a hierarchical scheme of sampled intermediate values.

Lemma 3 *(Bidirectional synchronisation)* Let $c_{next} \in \Sigma$ be a character, *cs* be a pattern over Σ and cs^{rev} be *cs* in reverse order. Given the intervals of the bidirectional FM-Index $[l, r)$, $[l^{rev}, r^{rev})$ and $[l^{rev}_{next}, r^{rev}_{next})$ that contain all rotations starting with *cs*, cs^{rev} and $c_{next}\ cs^{rev}$, respectively, the following formulae can be used to derive the interval $[l_{next}, r_{next})$ that contains all rotations starting with the pattern $cs\ c_{next}$:

$$\begin{aligned} l_{next} &= l + \mathit{Prefix\text{-}Occ}^{rev}[c_{next}, r^{rev}] - \mathit{Prefix\text{-}Occ}^{rev}[c_{next}, l^{rev}] \\ r_{next} &= l_{next} + r^{rev}_{next} - l^{rev}_{next} \end{aligned}$$

A visualization of the bidirectional synchronization can be observed in Figure 8.

Proof Firstly, it can be observed that the interval $[l, r)$ has to contain the interval $[l_{next}, r_{next})$, because all rotations starting with the pattern $cs\ c_{next}$ also start with the pattern *cs*. To determine l_{next}, the number of rotations in $[l, r)$ containing a lexicographically smaller character than c_{next} after *cs* has to be calculated and added to l. Simply adding this offset is sufficient, because the rotations in $[l, r)$ are tied in their ordering until the end of their starting pattern *cs*. Therefore, they are effectively sorted according to the characters after *cs*, the first of which being the characters that are compared to c_{next}. The data structures for the reverse t can be used to efficiently obtain the offset to add to l, because the characters behind the pattern *cs* in $[l, r)$ are exactly the characters of BWT^{rev}_t in the interval $[l^{rev}, r^{rev})$, just in a different order. Since the part of the above formula $\mathit{Prefix\text{-}Occ}^{rev}[c_{next}, r^{rev}] - \mathit{Prefix\text{-}Occ}^{rev}[c_{next}, l^{rev}]$ counts the number of lexicographically smaller than c_{next} characters within $[l^{rev}, r^{rev})$, this value is exactly the offset to add to l to obtain l_{next}.

To determine r_{next}, the size of the interval $[l^{rev}_{next}, r^{rev}_{next})$ is added to l_{next}, because the sizes of the intervals $[l_{next}, r_{next})$ and $[l^{rev}_{next}, r^{rev}_{next})$ have to be equivalent. It follows that the above formulae are a correct way to derive $[l_{next}, r_{next})$ from the given intervals of the bidirectional FM-Index. □

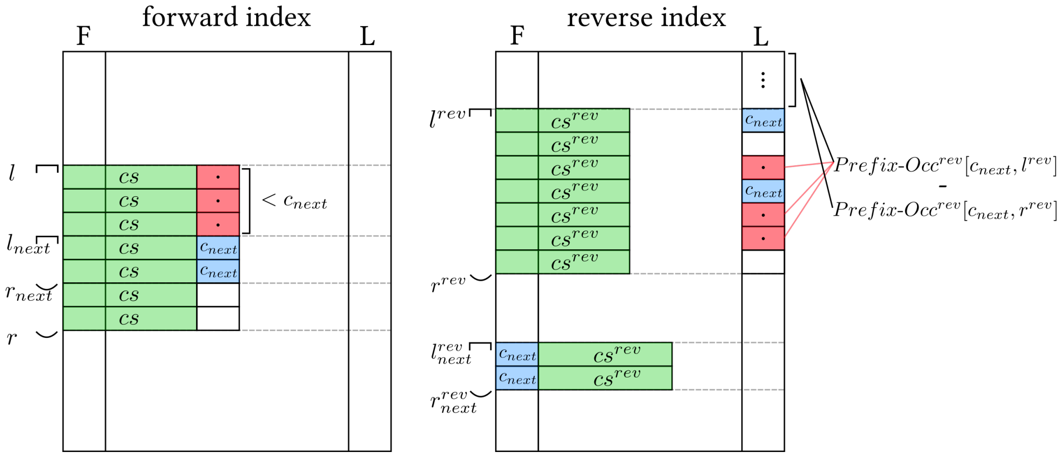

Figure 8: The synchronization step of a bidirectional FM-Index. Occurrences of the character c_{next} are highlighted in blue and occurrences of characters smaller than c_{next} are highlighted in red. The patterns *cs* and cs^{rev} are shown in green. The rightmost part of the figure shows how $\mathit{Prefix\text{-}Occ}^{rev}$ queries are used to calculate the offset to add to l in order to obtain l_{next}.

The EPR-dictionaries, which form the basis of the bidirectional FM-Index implementation used in this work, naturally support *Prefix-Occ* queries. Given the $(j-1)$-th and j-th smallest non-sentinel characters $c_{j-1}, c_j \in \Sigma$, they even implement *Occ* queries by computing $Occ[c_j, i] = \mathit{Prefix\text{-}Occ}[c_j, i] - \mathit{Prefix\text{-}Occ}[c_{j-1}, i]$.

2.3 Approximate Search with the FM-Index

2.3.1 Backtracking Search

Even without the capabilities of the bidirectional FM-Index, it is possible to modify the backwards search of the FM-Index to find all occurrences of the pattern in the text with up to k errors, rather than only exact matches. This can be achieved using a backtracking approach, as seen in Algorithm 3. At each step of the search, if the number of remaining errors is greater than zero, the algorithm branches for each non-sentinel character of the alphabet. The pseudocode depicts the version of the procedure that allows for up to k substitutions of characters, i.e. a Hamming distance of up to k. Additional support for indels, i.e. an edit distance of up to k, can be added by incorporating more branches in the procedure that take into account the possibility of insertions or deletions.

Algorithm 3: A backtracking algorithm to search the pattern p in a text of length n with a Hamming distance of up to k, using the FM-Index

```
FmIndexBacktracking ( $p = p[1]...p[m]$ , $k$ ) :
    SearchStepBacktracking ( $p$ , $k$ , $[1, n+1)$ )

SearchStepBacktracking ( $p = p[1]...p[i]$ , $k$ , curr_interval ) :
    if curr_interval is empty :
        return
    else if $k$ == 0 :
        Search $p$ using normal backwards search initialized with curr_interval
    else if $i == 1$ :
        Report occurrences from curr_interval
    else :
        for $c$ in $\Sigma \setminus \{\$\}$ :
            let next_interval = Pattern-Extension ( curr_interval , $c$ )
            if $c \neq p[i]$ :
                SearchStepBacktracking ( $p[1]...p[i-1]$ , $k-1$ , next_interval )
            else :
                SearchStepBacktracking ( $p[1]...p[i-1]$ , $k$ , next_interval )
```

The running time of Algorithm 3 can be analyzed as follows: there are $\binom{m}{k}$ ways of distributing the k errors along the pattern of length m. A way of distributing the errors along the pattern is called *error configuration* throughout this work. For every such error configuration, any character from Σ except for the sentinel and the actual character of p

at that position could be placed at every error slot. Thus, there are $O\left(\binom{m}{k} \cdot \sigma^k\right)$ leaves of the implicit search tree representing the execution of this algorithm. For every leaf, there are $m - k$ pattern extension steps in the worst case. It follows that the total running time of the algorithm is a function in $O\left(\binom{m}{k} \cdot \sigma^k \cdot m\right)$. The branches of the search tree in which less than k errors are distributed along the pattern were not included in this analysis, but counting them would lead to adding the same formula as above with a smaller k. They can therefore be omitted.

It follows that the backtracking approach is an extremely costly procedure, compared to the normal backwards search with a running time in $O(m)$, especially for larger values of k. In practice, backtracking is only considered feasible in the case of a single error, i.e. $k = 1$. However, it is possible to search with a larger number of errors by leveraging the capabilities of the bidirectional index and structuring the search more efficiently. The main idea is to branch for errors as late in the tree as possible. For example, for error configurations where the errors are clustered at the beginning and the end of the pattern, it would be advantageous to start the search in the error-free middle part of the pattern (see Figure 9, I). This has two advantages. Firstly, the number of operations required is reduced in the worst case, because large parts of the pattern are searched without branching in $O(m)$ time, or with only little branching. Secondly, there is a higher probability of the search stopping due to the absence of occurrences in the current interval (Algorithm 3, lines 4-5) before entering the expensive part of the search tree. When restructuring and optimizing the search in this way, it si important to make sure that intermediate intervals from previous searches are reused as much as possible, and that the whole space of possible error configurations is still covered.

2.3.2 Search Schemes

Search schemes [38] are a way of formalizing this concept of restructuring the backtracking search tree. First, the pattern is partitioned into l pieces, denoted by $P_1, ..., P_l$. It is not known what is the best strategy for setting l, but setting it to $k + 2$ has been to have advantages over setting it to smaller values [38]. Then, a search scheme is a set of searches $S = \left\{s_1, ..., s_{|S|}\right\}$. In each search, the pieces of the pattern are searched in a specific order using the bidirectional FM-Index, with given lower and upper bounds on the number of allowed errors per piece. Formally, a search is defined as a triplet (π_i, L_i, U_i) and can be interpreted as an individual search tree. π_i is a permutation of $\{1, ..., l\}$ and denotes the order in which the pieces of p are searched in s_i. To allow a seamless search using the bidirectional FM-Index, the permutation has to satisfy the so-called *connectivity property*. This property states that an index of a pattern piece can only appear in a position of π_i other than the first if it is smaller by 1 than the minium or larger by 1 than the maximum of all previous entries of π_i.

L_i and U_i are sequences of length l containing the lower and upper bounds on the number of accumulated errors in the search until the part of p defined by the corresponding entry of π_i is searched. The combination of upper and lower bounds can significantly limit the amount of branching per search of the search scheme. It therefore aids in making the whole search process more efficient. For example, given $l = 3, \pi_1 = (2, 3, 1), L_1 = (0, 1, 1), U_1 = (0, 1, 2)$ the search would would start with P_2 and exactly zero allowed errors. Then, the index would extend the currently searched part of the pattern by adding P_3 and while doing so, it would branch such that exactly one error is found in that part

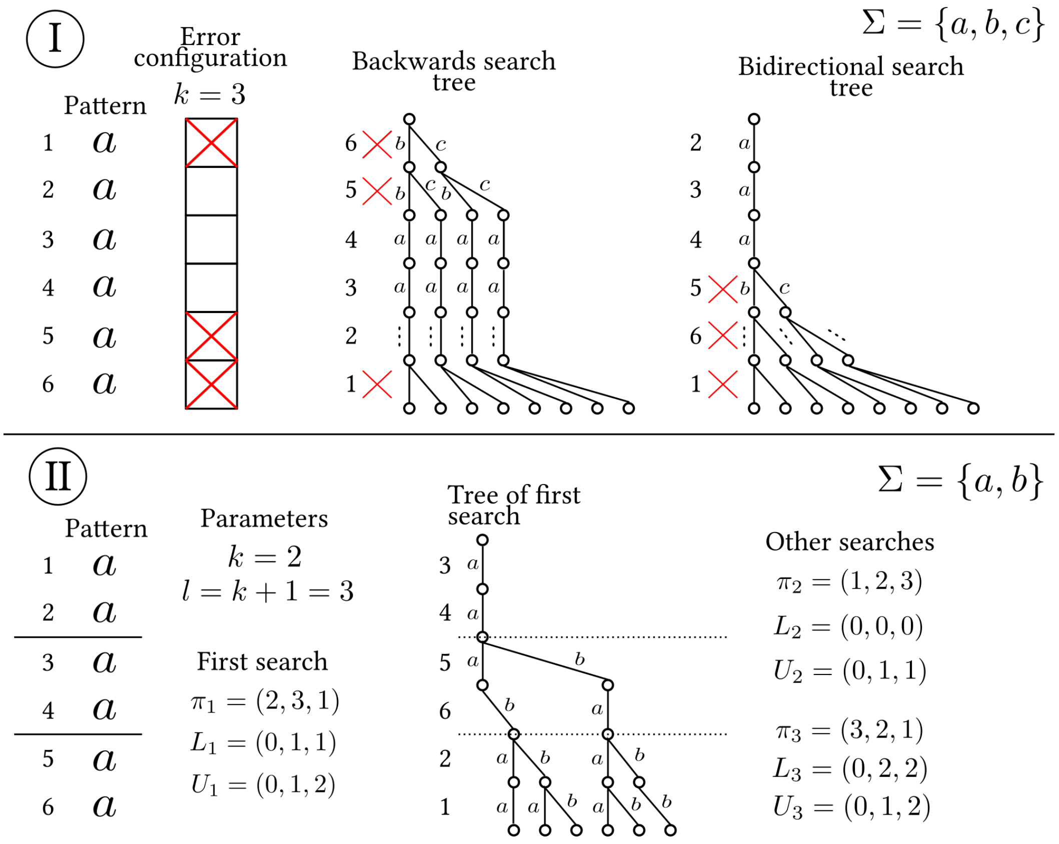

Figure 9: The two subfigures I and II show two different concepts related to searching with bidirectional FM-Indices. Subfigure I shows different orders in which the characters of the pattern *aaaaaa* could be searched with three errors and a given error configuration. The left tree shows the backwards search order and the right tree shows an order that could be searched with a bidirectional FM-Index. The branching in the right tree happens later, resulting in fewer nodes in the tree. Subfigure II shows how the text *aaaaaa* is partitioned into three pieces and provides an example of a complete, optimal search scheme for the given parameters. The tree in the middle shows how the upper and lower bounds limit the amount of branching that happens during the first search. It should be noted that this tree is conceptually different from the trees in subfigure I, because it is not limited by a single error configuration.

of the pattern. Finally, the currently searched part of the pattern would be extended by adding P_1 to the front. In this last part, zero or one errors would be allowed, such that the total accumulated number of errors is one or two. A visualization of this example can be found in Figure 9, part II.

L_i and U_i have to be monotonously increasing and at every position, U_i has to be greater than or equal to the corresponding position of L_i. If the searches in S cover all possible configurations in which the given number of errors can be distributed among the pattern pieces, then S is called *complete*. Complete search schemes can be used to significantly speed up the search of patterns with errors using the bidirectional FM-Index. If every error configuration is visited at most once across all of the searches, the search scheme is called *non-redundant*. Complete, non-redundant search schemes using the Hamming distance metric are guaranteed to report each occurrence of the pattern exactly once.

However, edit distance based search schemes can always report occurrences multiple times, because the same occurrence can be matched using different configurations of the edit distance operations.

The quality of search schemes can be graded by the number of required pattern extensions to execute all searches, called the *node count*. Several heuristic approaches exist that attempt to compute search schemes with low node counts. For the Hamming distance and small values of k and l, it is even possible to compute search schemes with optimal node counts by formulating the problem as a mixed integer linear program [39], [40]. These *optimum search schemes* have been shown to also work well for searching with errors in edit distance, and are used for searching patterns with $k \in \{1, 2, 3\}$ and $l = k + 2$ in this work.

Recently, an alternative metric for grading the quality of search schemes has been proposed, called *weighted node count* [41]. It aims to more accurately predict the performance of search schemes by weighting nodes according to the probability of being visited during a search. Nodes that are located deeper in the search tree are assigned a smaller weight. This innovation may lead to the design of more efficient search schemes in the future, but is not considered further in this work.

2.4 Interval Tree

Interval trees are data structures for storing sets of numeric intervals. They allow the retrieval of all intervals that overlap a given target interval more efficiently than storing them in a list or conventional tree structure would. If the interval tree stores n intervals and K intervals overlap with the target interval, the K overlapping intervals can be found in $O(log\, n + K)$ time [42]. This data structure is not the focus of this work and is only used as a black box. Therefore, the implementation details are not discussed here, but can be found in [42].

3 Methods

This chapter describes in detail the new algorithmic ideas developed in this work to implement the prototype long-read mapper, which mostly uses exact algorithmic components. First, the main ideas of this work on how to adapt PEX algorithm for long-read mapping are presented. Then, a novel bottom-up PEX tree construction algorithm is introduced that allows for a more robust implementation of these ideas. The final two sections of the chapter discuss several algorithmic concepts that have helped to improve the prototype read mapper of this work.

3.1 Seeds with Allowed Errors in the PEX Algorithm

The first main idea of this work is to apply the PEX algorithm (see Section 2.1) to long-read mapping and search the pigeonhole seeds individually with the FM-Index (see Section 2.2) instead of using a multipattern search algorithm. This is advantageous, because even though multipattern search algorithms could theoretically search all reads at once, it is not practically feasible to search millions of patterns with them in genomes consisting of billions of base pairs.

However, directly applying the PEX algorithm may not work well, because practically relevant long-read error rates of around 8% would lead to seeds of length around 12 in the leaves of the PEX tree. In sequences as long as typical genomes, seeds of such short lengths can often be found many times by random chance only and therefore are not suitable for read mapping. Figure 10 shows the average number of occurrences per seed in the human genome of one million randomly generated seeds with different lengths and numbers of errors (including indels)[4]. Seeds of length 12 with 0 errors occurred on average about 6 times. Seeds in real applications are sampled from the space of biological sequences and not from the space of all possible strings. It is therefore expected that even more random matches occur. It follows that PEX seeds with 0 errors are unsuitable for use in seed-extend based read mapping applications.

To solve this issue, the second main idea of this work is to change the seeds of the PEX algorithm to be longer, by also allowing for errors in the search. For an error rate of 8%, the PEX tree nodes adjacent to the leaves are associated with one or two errors and pattern subsequences of lengths around 24 or 36, respectively (see Figure 11 for an example PEX tree). In the experiment of Figure 10, random sequences of length 24 searched with 1 error occurred on average less than 0.0001 times in the human genome and random sequences of length 36 searched with 2 errors never occurred. It follows that skipping most of the leaves and directly searching the sequences in the adjacent nodes with their associated number of errors is suitable for mapping long-reads with 8% error rate in the human genome. In Figure 11, the nodes that could be skipped are outlined in red and the new leaves have a green outline. For data with even higher error rates or data sets consisting

[4]The number of occurrences for more than 0 errors may be inflated due to occurrences with different error configurations matching at the same positions in the genome. However, this effect does not take away from the interpretation of this figure. More details on this can be found in Section 3.4.1.

of multiple genomes, it may be necessary to skip more nodes than just the leaves, so that the seed sequences are searched with 2 or more errors.

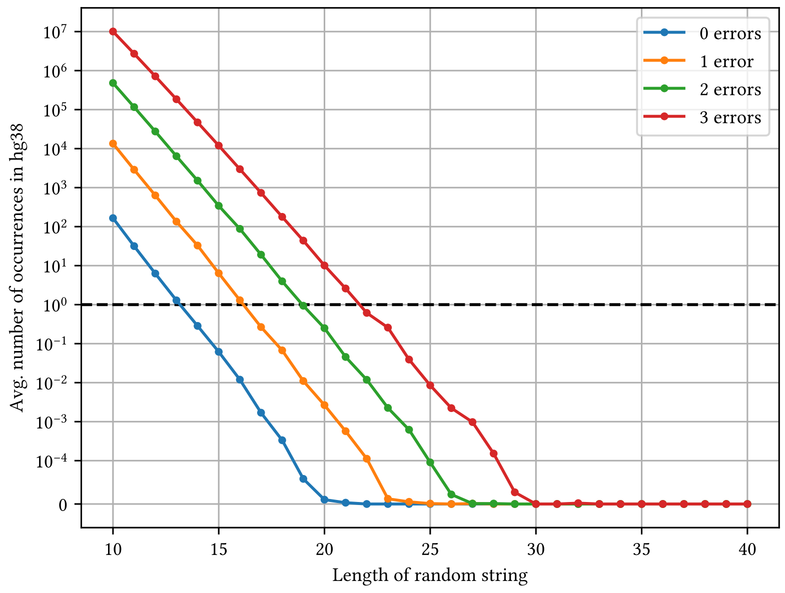

Figure 10: The average number of occurrences in the hg38 human genome assembly of one million randomly generated seed sequences, distinguished by length (x-axis) and number of errors (including indels)[4]. The x-axis has a logarithmic scale. The value of one occurrence on average is highlighted by a dashed horizontal line.

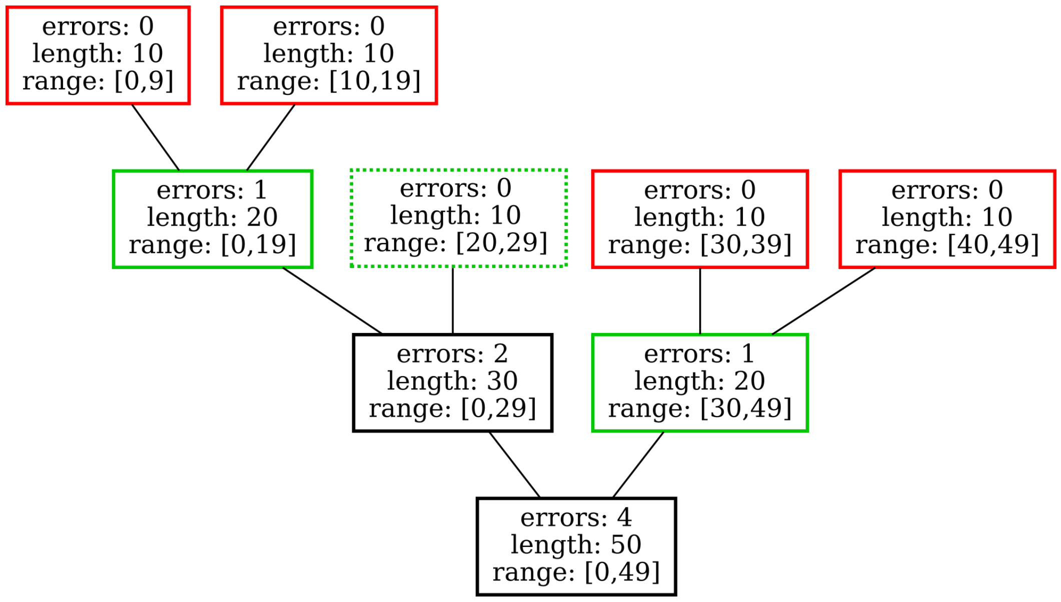

Figure 11: The PEX tree for a pattern of length 50 and an error rate of 8% (4 errors). The old leaves, which are not searched in the modified PEX algorithm of this work, have a red outline. The new leaves, most of which produce seeds more suitable for long-read mapping with high error rates, have green outlines. Instead of keeping the leaf with the dotted outline, it would be possible to remove it and the other child of its parent node. Keeping it would correspond to the result of Algorithm 4 with s set to 1.

Some care needs to be exerted when implementing this idea. Simply removing all nodes with an error count less than a certain value from the original PEX tree is not a sound approach. If only one of the two children of a parent node were removed in this way, the generalized pigeonhole lemma would no longer hold in that part of the tree. In Figure 11, the node with the dotted outline is not removed for this reason.

To correctly implement this change, the PEX tree construction algorithm has to be modified to stop the recursive tree expansion earlier. This can be done by adding a new parameter s that controls the maximum number of errors in the leaves of the tree (see Algorithm 4, lines 1-3) and adjusting the base case condition for the recursive tree generation function (line 5). Using this simple approach, it is not possible to guarantee that all leaves of the resulting PEX tree are associated with exactly s errors. Leaves with less than s errors can appear, because not all numbers of errors are visited on every path from the root of the PEX tree to the leaves. For example, in Figure 11, the node with the dotted outline would be kept in the case of $s = 1$, because the number of errors jumps from 2 in the parent node to 0 in the leaf.

Algorithm 4: Modified PEX tree construction for pattern p, k errors and at most s errors in the leaves. Modifed parts in red.

```
1 ConstructTree ( $p = p[1]...p[m]$ , $k$ , $s$ ) :
2     AddNodes ( $p$ , $k$ , $\perp$ , $\left\lfloor \frac{m}{k+1} \right\rfloor$ , $s$ )

3 AddNodes ( $p = p[i]...p[j]$ , $k$ , $parent\_node$ , $piece\_length$ , $s$ )
4     Add node $curr\_node$ to tree with parent $parent\_node$, pattern range $[i, j]$ and $k$ errors
5     if $k \leq s$ :
6         Mark $curr\_node$ as leaf
7     else :
8         Keep expanding tree (same as Algorithm 1)
```

The prototype read mapper of this work applies the bidirectional FM-Index (see Section 2.2.1) and search schemes (see Section 2.3.2) to search the longer sequences of the new leaves directly with their associated numbers of errors, instead of searching the leaves of the original PEX tree with 0 errors. The exactness guarantees of the PEX algorithm are upheld, because complete (optimum) search schemes are used. Therefore, directly searching the sequences of the new leaves is equivalent to searching the leaves of the original tree and performing hierarchical verification until the inner node, which has become a leaf in the new tree, is reached. Furthermore, the original way of showing the exactness of the PEX algorithm by recursively applying the generalized pigeonhole lemma still works in exactly the same way.

3.2 Bottom-Up PEX Tree Construction

The approach shown in Algorithm 4 is not the optimal solution to the goal of creating PEX trees that have leaves associated with more than 0 errors. As can be observed in

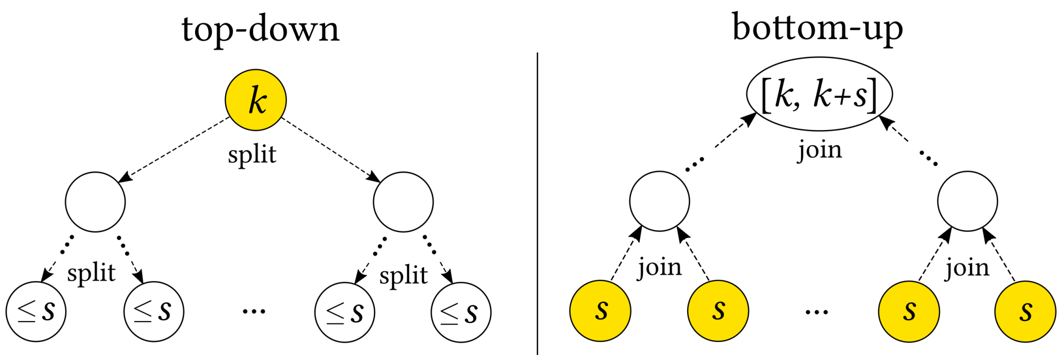

Figure 12: A comparison between the bottom-up and top-down approaches to PEX tree construction. The nodes that are part of the initialization of the respective algorithm are marked in yellow. The top-down approach initializes a root with k errors and repeatedly splits nodes it until leaves with at most s errors are obtained. On the other hand, the bottom-up algorithm is initialized with a specific number of leaves that are associated with exactly s errors. These leaves are merged repeatedly, until only a single root remains. The analysis in the remainder of this section will show that the number of errors of the resulting root is in the range $[k, k + s]$.

Figure 11, if $s = 1$, a leaf with 0 errors would be kept in the tree. Such a leaf could produce a large number of random, false positive anchors. With $s = 2$, the leaf with 0 errors would be removed, but leaves with 2 errors would of course appear. This could be detrimental to the running time, as searching with 2 errors is significantly more expensive than searching with 1. Even the use of optimum search schemes cannot compensate for the fact that the number of errors exponentially increases the running time of the FM-Index approximate search. Since Figure 10 showed that PEX seeds with 1 error would lead to an acceptably low number of random occurrences for long-read error rates of around 8%, it is likely more efficient to only search seeds with 1 error instead of with 2.

To solve this problem and allow precise control over the number of errors in the PEX tree leaves, this work proposes a different way of constructing the PEX trees, called *bottom-up PEX tree construction*. In short, it works by creating leaves with an error count of exactly s and iteratively merging nodes using formulae derived from the generalized pigeonhole lemma (Lemma 1). The following sections will first explain how these formulae are derived and then describe the bottom-up PEX tree construction algorithm in detail. Figure 12 compares the bottom-up and top-down approaches to PEX tree construction.

3.2.1 Determining the Number of Leaves

As a reminder, the generalized pigeonhole lemma states the following:

Lemma (*Generalized pigeonhole lemma, applied to approximate string matching*) Assume there exists an occurrence *occ* in the text of a pattern p with k errors (including indels). Let $P = P_1, \ldots, P_l$ be a partition of the pattern into subpieces and $a_1, \ldots, a_l$ be non-negative integers such that $A = \sum_{i=1}^{l} a_i$. Then, for at least one $i \in \{1, \ldots, l\}$, the occurrence of p in the text must contain a substring that matches P_i with at most $\left\lfloor \frac{a_i \cdot k}{A} \right\rfloor$ errors.

A useful interpretation of the value A is a minimum number of errors between a part of the text and the pattern p. If this minimum is met or exceeded, the part of the text is not in the result set of the search algorithm. Similarly, the weight a_i can be seen as a minimum number of errors between a part of the text and the i-th subpiece. If this minimum is met or exceeded, the part of the text is not in the set of candidate regions to be considered for an occurrence of the whole pattern, anchored at the i-th subpiece. When setting $A = k + 1$ and $\forall i \in \{1, ..., l\} : a_i = 1$ according to this interpretation, it follows that l can be set to $k + 1$ to make the basic pigeonhole seeding logic appear:

$$\forall i \in \{1, ..., l\} : \left\lfloor \frac{a_i \cdot k}{A} \right\rfloor = \left\lfloor 1 \cdot \frac{k}{k+1} \right\rfloor = 0$$

In words, p has to match with less than $k + 1$ errors and is partitioned into $k + 1$ pieces. At least one of these pieces has to match a substring of *occ* exactly. To allow for leaves with more than 0 errors in the PEX trees, the parameters have to be set in the following way. $A = k + 1$ remains the same, because the whole pattern in the root of the tree still needs to match with k errors. However, the a_i weights need to be set to $s + 1$ to ensure that at least one piece i of the pattern matches with at most s errors:

$$\left\lfloor \frac{a_i \cdot k}{A} \right\rfloor = \left\lfloor (s+1) \cdot \frac{k}{k+1} \right\rfloor \leq s$$

The sum of the weights a_i has to equal A and all a_i have the same value. Therefore, the following formula for l is obtained:

$$A = a_i \cdot l \iff l = \frac{k+1}{s+1}$$

In this context, the meaning of l is the number of leaf nodes to be created in the initialization of the bottom-up PEX tree construction algorithm. Since it has to be integer, the ceiling function has to be applied to the rightmost term of the above equation. The value for A is slightly increased by this change, but s remains an upper bound on the number of errors of each subpiece. Adding the ceiling function has the negative effect that the algorithm might search for occurrences of p with up to $k + s$ errors, instead of only k. A reasonable way to avoid this side effect would be to set the number of associated errors of $s - 1$ leaves to $s - 1$. However, this change could lead to the introduction of leaves that do not have the desired error count of s and lead to a large number of random, false-positive anchors. In addition, the change from k to $k + s$ errors is negligible for the application of mapping long-reads with high error rates, and unwanted pattern occurrences could easily be filtered out in a post-processing step. Therefore, the bottom-up PEX algorithm of this work is allowed to search for pattern occurrences with up to $k + s$ errors.

In summary, the previous paragraphs have shown how to generalize the pigeonhole seeding strategy for seeds with more than 0 errors. The result is used in the initialization of the bottom-up PEX tree construction algorithm. In this phase, $\left\lceil \frac{k+1}{s+1} \right\rceil$ leaves are created, each associated with s errors.

3.2.2 Iteratively Adding Parent Nodes

To take the next step and create a PEX tree for the hierarchical verification, the following lemma shows how to create a parent node for two or more PEX tree nodes with adjacent pattern ranges. The pattern range of the parent node has to be the union of the pattern ranges of the children.

Lemma 4 *(Number of errors of bottom-up PEX tree parent nodes)* Given $l_{child} > 1$ child nodes with the associated numbers of errors $k_1, ..., k_{l_{child}}$, the number of errors of the parent node k_{parent} is set to the following value:

$$k_{parent} = l_{child} - 1 + \sum_{i=1}^{l_{child}} k_i$$

When k_{parent} is set in this way, the generalized pigeonhole lemma upholds between the parent and its children and allows for the subtree rooted at the newly created parent to be used in the hierarchical verification.

Proof The weights a_i of the generalized pigeonhole lemma for the children are set to $k_i + 1$. It follows that $A = l_{child} + \sum_{i=1}^{l_{child}} k_i$. The lemma states that if there exists an occurrence of the pattern range of the parent with at most k_{parent} errors, then there exists a child node i whose sequence range matches a substring of the parent's match with at most k_i errors:

$$\left\lfloor a_i \cdot \frac{k_{parent}}{A} \right\rfloor = \left\lfloor (k_i + 1) \cdot \frac{l_{child} - 1 + \sum_{i=1}^{l_{child}} k_i}{l_{child} + \sum_{i=1}^{l_{child}} k_i} \right\rfloor = \left\lfloor (k_i + 1) \cdot \frac{k_{parent}}{k_{parent} + 1} \right\rfloor \leq k_i$$

This bound ensures that the subtree rooted at the newly created parent can be used in the hierarchical verification. □

Using this way of setting the number of errors of the parent nodes, it is possible to construct a PEX tree by creating a set of leaves and iteratively adding nodes in arbitrary ways. However, it is still preferable to construct a (mostly) binary tree, because a high amount of branching would lead to a flat tree, which would reduce the effectiveness of the hierarchical verification.

3.2.3 Guaranteeing the Parameters of the Root Node

There is one last issue that needs to be resolved before the bottom-up PEX tree construction can be considered a valid approach. So far it is not clear that the proposed way of creating leaves and adding parents in arbitrary ways leads to a PEX tree in which the root is associated with a number of errors in the range $[k, k + s]$[5]. The following theorem states that this is indeed the case.

Theorem 1 *(Bottom-up PEX tree construction)* Let S be a set of PEX trees, initially containing $\left\lceil \frac{k+1}{s+1} \right\rceil$ single-node trees, each associated with s errors. Lemma 4 can be applied to add a new tree to S by removing and merging two or more trees from the set with a parent node. If this is done repeatedly until only a single tree is left, the root of the resulting tree is associated with a number of errors in the range $[k, k + s]$.

Proof The proof is conducted via induction over the number n of parent nodes that are added while creating the tree, excluding the final parent node that becomes the root

[5]The reason for using the range $[k, k + s]$ instead of just k is the fact that $k + 1$ may not be divisible by $s + 1$. In Section 3.2.1, this limitation of the algorithm is described in more detail.

of the final PEX tree. In the following, the value k_i^j refers to the associated number of errors of the i-th remaining PEX tree root *after* the addition of the j-th parent root, with k_i^0 being the initial weights of the leaves. L_j refers to the number of PEX trees remaining in the forest *after* the j-th parent node was added, with $L_0 = l$ being the initial number of leaves. The value l_j refers to the number of PEX trees merged by the j-th creation of a parent node.

Induction start: $(n = 0)$

In this case, no PEX tree nodes are added to the tree except for a single root connecting all leaves. It follows that $L_0 = l_1 = l$ and $\forall i \in \{1, ..., l\} : k_i^0 = s$. The following bounds for the number of errors of the root k_{root} are obtained:

$$\begin{aligned}
k_{root} &= L_0 - 1 + \sum_{i=1}^{L_0} k_i^0 \\
&= l - 1 + l \cdot s \\
&= l \cdot (s+1) - 1 \\
&= \left\lceil \frac{k+1}{s+1} \right\rceil \cdot (s+1) - 1 \\
&\geq \frac{k+1}{s+1} \cdot (s+1) - 1 \\
&= k + 1 - 1 \\
&= k
\end{aligned}$$

$$\begin{aligned}
k_{root} &= \left\lceil \frac{k+1}{s+1} \right\rceil \cdot (s+1) - 1 \\
&< \left(\frac{k+1}{s+1} + 1 \right) \cdot (s+1) - 1 \\
&= \frac{k+1}{s+1} \cdot (s+1) + (s+1) - 1 \\
&= k + 1 + s + 1 - 1 \\
&= k + s + 1
\end{aligned}$$

Since k_{root} has to be integer, it follows that it is in the range $[k, k+s]$. As a side note, the base case of this induction also works by construction, since it is equivalent to the generalized pigeonhole seeding logic used above to the derive the formula for the number of leaves to create.

Induction step: $(n \to n + 1)$

An example that visualizes the arguments in this part of the proof can be found in Figure 13. The goal is to show that the following value is in the range $[k, k+s]$:

$$k_{root} = L_{n+1} - 1 + \sum_{i=1}^{L_{n+1}} k_i^{n+1}$$

It refers to the number of errors k_{root} in the final PEX tree root, which connects all trees of the forest remaining after adding $n + 1$ parent nodes. From the induction hypothesis it is known that the same value after adding n parent nodes is in the range $[k, k + s]$. Here it is be referred to as ${k_{root}}'$:

$$ {k_{root}}' = L_n - 1 + \sum_{i=1}^{L_n} k_i^n $$

The following two observations suffice to show that $k_{root} = {k_{root}}'$. Firstly, the number of trees in the forest after adding $n + 1$ parent nodes L_{n+1} is equal to $L_n - l_{n+1} + 1$, because l_{n+1} trees are merged into a single root. In the right part of Figure 13, 2 out of the 4 initially remaining roots have been merged in step 2, resulting in $4 - 2 + 1 = 3$ roots after the operation. Secondly, the following equation holds between the sum of errors in the remaining roots after n and $n + 1$ added parent nodes:

$$ \sum_{i=1}^{L_{n+1}} k_i^{n+1} = \sum_{i=1}^{L_n} k_i^n + l_{n+1} - 1 $$

This is true, because the sum on the right hand side of the equation can be conceptually split into the PEX tree roots being merged into the $(n + 1)$-th parent node and those not part of that operation. The sum of the ones part of the operation added to $l_{n+1} - 1$ is exactly the weight of the new node added by the operation, according to Lemma 4. It follows that the right hand side is equivalent to the sum of errors of roots not merged into the $(n + 1)$-th parent node plus the number of errors of the newly created parent node. This is exactly the term on the left hand side of the expression.

In Figure 13, the sum of errors of the initially remaining roots in step 1 is $3 + 3 + 2 + 2 = 10$. In the right part of the image, 2 of the initially remaining roots are merged in step 2. It follows that the right side of the above equation is equal to $10 + 2 - 1 = 11$ in this example. The left side of the equation also equals 11, because the weight of the new node is 7 and two roots with weights of 2 were left untouched.

Using these two observations, it can be simply shown that $k_{root} = {k_{root}}'$:

$$ \begin{aligned} k_{root} &= L_{n+1} - 1 + \sum_{i=1}^{L_{n+1}} k_i^{n+1} \\ &= L_n - l_{n+1} + 1 - 1 + \sum_{i=1}^{L_{n+1}} k_i^{n+1} \\ &= L_n - l_{n+1} + \sum_{i=1}^{L_n} k_i^n + l_{n+1} - 1 \\ &= L_n + \sum_{i=1}^{L_n} k_i^n - 1 \\ &= {k_{root}}' \end{aligned} $$

□

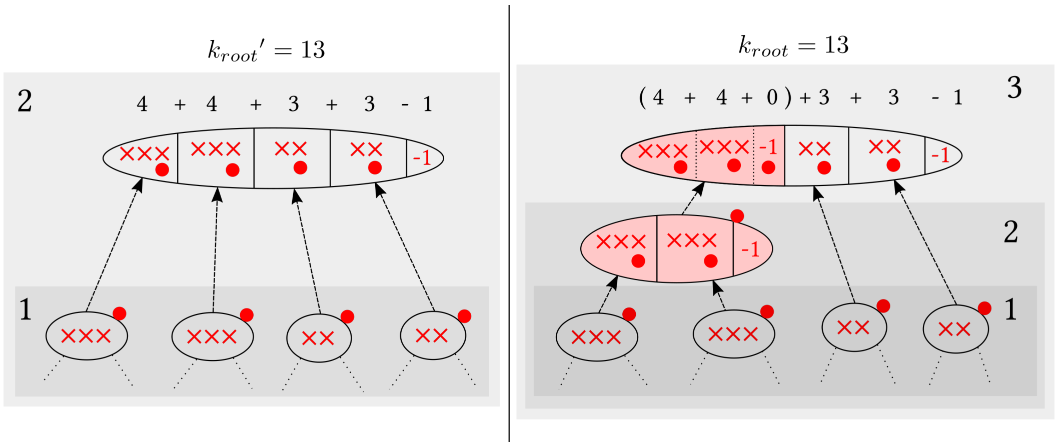

Figure 13: A visualization of the inductive step of the proof of Theorem 1. In both the left and the right part of the image, 4 PEX tree roots remain initially (1). In the left part, which can be understood as the situation described by the induction hypothesis, all of these roots are then merged into one final root (2, left). In the right part, step 2 instead consists of merging 2 of the initially remaining roots. The newly created root has a red background. Afterwards, the 3 still remaining roots are merged into the final root (3). The errors associated with the roots of step 1 are visualized by red crosses. The roots that are merged in later steps carry a red dot that symbolizes the +1 value they contribute to the error count of their parents, apart from their own number of errors. When these roots are merged, the red dot is shown inside their parents and also represents an error. It can be observed that the additional operation on the right side of the figure does not alter the number of errors in the final root of the tree, because the additional red dot is offset by an additional −1 value.

With the result of Theorem 1, the approach of constructing PEX trees in a bottom-up way can be considered valid. Algorithm 5 shows the detailed pseudocode of the procedure. Lines 2-4 show the initialization of the algorithm. It creates l leaves that are associated with s errors. In the implementation, the PEX trees are stored in a sequence to ensure that only trees with roots that have adjacent pattern ranges are merged. Lines 5-9 show the main loop of the algorithm. In each iteration, one level of the tree is added by merging pairs of adjacent subtrees with a new parent node. The parameters of the parent node are set according to Lemma 4 (lines 14-18). To simplify the procedure, three subtrees are merged if a level contains an odd number of nodes (line 12). This may slightly reduce the effectiveness of the hierarchical verification in rare edge cases, but a trade-off for simplicity of the implementation is chosen in this work.

Algorithm 5: Bottom-up PEX tree construction for pattern p, k errors and at most s errors in the leaves

```
1  BottomUpConstructTree ( $p = p[1]...p[m]$ , $k$ , $s$ ) :
2      let $l$ = $\left\lceil \frac{k+1}{s+1} \right\rceil$
3      let ranges = $l$ pattern ranges that partition $p$ as evenly as possible
4      let curr_subtrees = sequence of PEX leaf nodes associated with $s$ errors, one for
       every pattern range from ranges

5      while curr_subtrees.length > 1:
6          let next_subtrees = empty sequence
7          for children in ChildGroupSequence ( curr_subtrees ) :
8              append CreateParent ( children ) to next_subtrees
9          curr_subtrees = next_subtrees

10     The final PEX tree is the remaining element of curr_subtrees.

11 ChildGroupSequence ( sequence ):
12     return sequence of adjacent pairs of the elements of sequence, starting with the
       first two elements. If the number of elements in sequence is odd, the last element
       is a triplet.

13 CreateParent ( children ):
14     let $l_{child}$ = children.length
15     let child_errors = sum of all associated errors of children
16     let $k_{parent}$ = $l_{child} - 1 + child\_errors$
17     let parent_pattern_range = union of pattern ranges of children
18     return new PEX node associated with $k_{parent}$ errors, the pattern range
       parent_pattern_range and the children children
```

3.3 Avoiding Repetitive Verification

In the textbook that serves as the reference description of the PEX algorithm for this work, it is mentioned that when using the pigeonhole seeding approach, some care has to exercised to report the verified pattern occurrences in the correct text order and to avoid reporting the same occurrence more than once [33]. The order of occurrences in the output is not relevant to read mapping, because different downstream applications require the data to be arranged in different ways and typically use software like `samtools` [43] to perform the necessary post-processing steps.

Deciding when two read alignments at neighboring positions in the reference genome should be considered different is a surprisingly involved task [21] and not the focus of this work. However, if multiple seeds of a read match at the same occurrence in the reference, it might be extremely wasteful to run a hierarchical verification for each one of them. Figure 14 shows how multiple anchors are found at the same read occurrence and lead to repetitive, unnecessary verifications. In practice, it is likely that more than one anchor is found per read occurrence, because the guarantee of the PEX algorithm's guarantee of finding at least one anchor per occurrence is based on the most unfavorable distribution of errors along the pattern. Such a distribution is unlikely to frequently occur in real data. Unnecessarily verifying multiple anchors for a single read occurrence therefore might have a significant negative impact on the running time performance when mapping long-reads, because single long-reads can generate thousands of pigeonhole seeds. Additionally, long-read occurrences are expensive to verify due to the high sequence length and the quadratic scaling of the verification algorithm.

To avoid this inefficiency, the prototype read mapper of this work keeps track of reference sequence intervals in which the presence or absence of a read occurrence has already been verified. The stored intervals always correspond to a verification run of the root node of the PEX tree of the current read. Then, before starting a hierarchical verification for an anchor, the algorithm checks whether it would lead to the verification

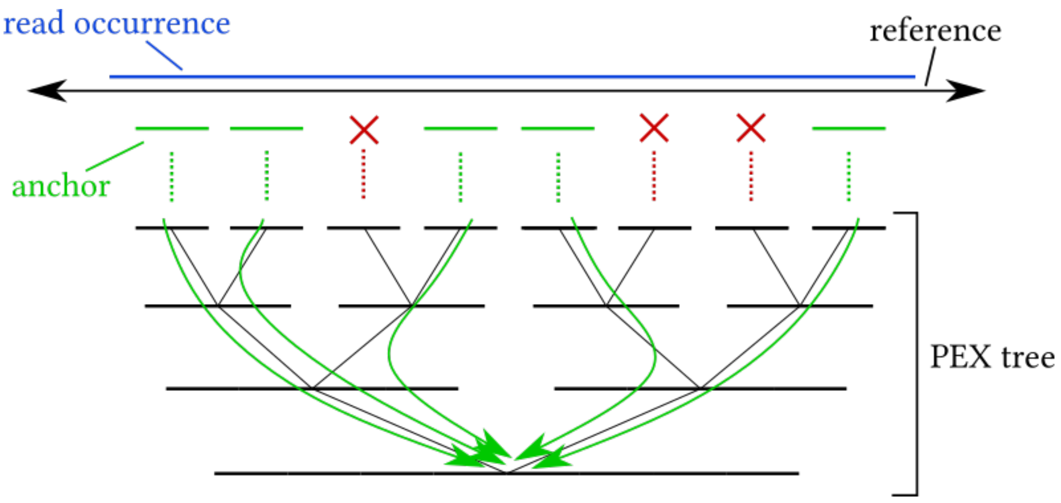

Figure 14: A read occurrence is shown in blue, next to the reference sequence in black. Below, there is a PEX tree with 8 leaves, 5 of which have produced an anchor in the read occurrence. In such a scenario, a naive implementation of the PEX algorithm would perform 5 hierarchical verifications, indicated by green arrows.

of a previously verified reference interval in the root of the PEX tree. If the interval has already been verified, the entire hierarchical verification procedure is skipped. This is a simple optimization that does not change the result set of the whole algorithm. It could be ineffective, because insertion or deletion errors could cause the verification interval to shift slightly for different anchors of the same occurrence. This could lead to the repetitive verification of almost completely overlapping intervals that lead to the same occurrence being found. However, excluding overlapping intervals from repetitive verification is not a suitable approach for this work as it could lead to the loss of results. If a read occurrence could not be verified in a first interval, there may still be an occurrence in a second interval that almost completely overlaps the first.

A possible way to improve this issue is to verify a slightly larger reference sequence interval than necessary in the last step of the hierarchical verification. This slightly larger interval is calculated by taking the original interval as defined by the PEX algorithm and adding a small amount of additional space on both sides. It is stored in the collection of previously verified intervals instead of the original interval. When looking up whether an interval has already been verified before starting a hierarchical verification, the original interval as defined by the PEX algorithm is used and not the one with extra space on both sides. The function that performs this look-up is changed from searching for the exact same interval to searching for a previously verified interval that contains the interval to be verified. To implement this idea, a new parameter r is introduced, which represents the ratio between the size of the original interval as defined by the PEX algorithm and the small extra space on either side of the slightly larger interval. For example, given $r = 5\%$ and an original interval size of 100, the slightly larger interval would have a size of $5 + 100 + 5 = 110$. The value of r should not be too large, because otherwise the additional cost of verifying larger sequence windows may negatively impact the running time performance and memory usage. A good value for r is determined empirically (see Section 4.7). It should also be noted that this optimization can change the result set of the algorithm, because only one of two almost completely overlapping occurrences may be reported. If this is unacceptable for a given downstream application, r could be set to zero to disable this behavior.

The data structure used to store the intervals could be a simple list, or an interval tree. The interval tree is more efficient in theory, but performance profiling of the long-read mapper prototype showed that the additional steps of storing and looking up the verified intervals do not have a significant impact on the running time of the program, regardless of the implementation. However, the prototype still uses interval trees to hedge against future changes of the algorithmic components that might alter the performance profile of the application.

3.4 Reducing the Number of Anchors

When the prototype long-read mapper of this work was first tested on a data set of real biological sequences, the program crashed due to an out-of-memory exception, which was likely caused by an enormous number of anchors generated by certain seeds. There were several reasons for this large number of anchors. This section describes these reasons and provides solutions to mitigate the issues they cause.

3.4.1 Removing Locally-Suboptimal Anchors

The first, less severe reason for the large number of anchors leading to the crash was that edit distance based search schemes can report the same occurrence of the pattern multiple times with different arrangements of the edit distance operations, i.e. different alignments. Some of these redundant anchors can be optimized away by FM-Index search implementations, but not necessarily all of them. Even the highly optimized implementation used in this work [44] reports multiple anchors for single seed occurrences. For example, when searching the pattern "AAAA" in the text "BBBAAAABBB" with 2 errors, 8 different anchors are found. It would be desirable to only obtain a single anchor that represents the match of "AAAA" with zero errors at the fourth position of the text.

To avoid unnecessary work when processing redundant anchors, the prototype read mapper of this work applies a simple filtering step after the anchors are retrieved. It sorts the retrieved anchors and removes all *locally-suboptimal* anchors. An anchor is locally-suboptimal, if it matches with k errors and another anchor matches i characters away with at most $k - i$ errors, where $0 < i \leq k$. Anchors that are not locally-suboptimal are called *locally-optimal*. If a locally-suboptimal anchor would lead to the successful verification of an occurrence, another anchor with fewer errors could lead to the verification of the same occurrence. The shift by i of the starting positions of the verification windows could be compensated by spending the surplus of i available errors as deletions of reference characters at one end of the verification window. It follows that locally-suboptimal anchors can be filtered out before the verification step without losing any results. A visualization of this argument can be observed in Figure 15.

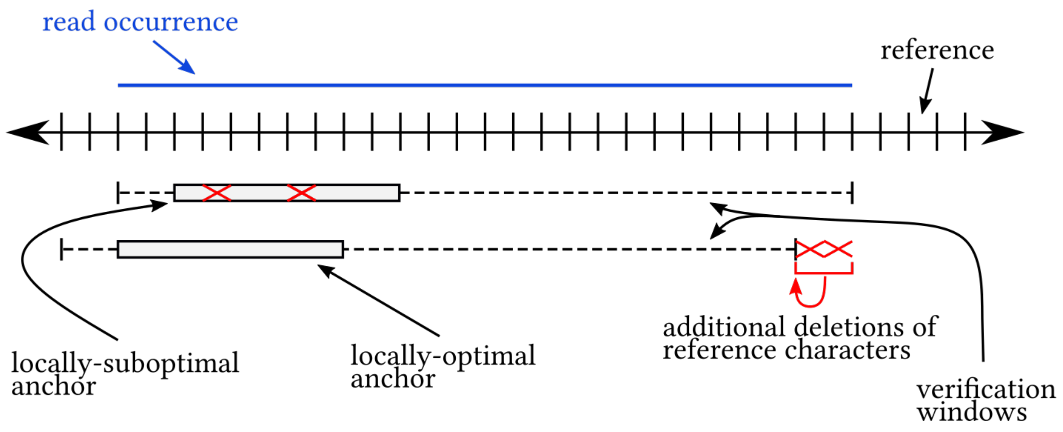
read occurrence
reference
locally-suboptimal anchor
locally-optimal anchor
additional deletions of reference characters
verification windows

Figure 15: A locally-suboptimal anchor with 2 errors and a locally-optimal anchor without errors. Both match inside of a read occurrence, but the verification window of the locally-suboptimal anchor covers more of the occurrence. However, by using the surplus of 2 errors as deletions of reference characters, the occurrence can also be detected in the verification window of the locally-optimal anchor.

3.4.2 Dealing with Repetitive Sequence

The optimization from the previous section can help reduce the number of anchors and avoid unnecessary work. However, it does not explain why the prototype read mapper crashed when first tested on real data. The main reason for the crash likely was that certain seeds matched in regions of repetitive sequence of the genome and produced a number of anchors that would be virtually impossible to end up with when searching seeds in a randomly generated sequence. For example, there were with a length of about 30 that were searched with two errors for which up to 5 million[6] anchors were found. The read with the highest total number of anchors for all of its seeds had almost 20 billion anchors[6]. It would be infeasible to run hierarchical verifications for all of these anchors. It follows that the prototype long-read mapper of this work has no choice but to exclude some of the anchors for seeds with large numbers of anchors, even if this leads to the possibility of excluding anchors that are located in a viable read occurrence. Excluding these anchors introduces the possibility of completely missing read occurrences, which breaks the exactness-guarantee of the algorithm.

A simple way of implementing this is to introduce an additional parameter called the *hard anchor cap*, which describes the maximum number of anchors a seed is allowed to have. If the number of anchors for a seed exceeds this threshold, the whole seed including all of its anchors is removed from further consideration. This decision is made after the `count` operation of the FM-Index, to avoid unnecessarily materializing large numbers of anchors in the `locate` operation.

The hard anchor cap allows the program to run on data sets of real biological sequences. However, it may lead to the exclusion of more anchors than necessary. Since the goal of the long-read mapper prototype of this work is to be as exact as possible, a less strict way of dealing with large numbers of anchors should be considered. One such approach is the *soft anchor cap*. It describes the maximum number of anchors per seed that are allowed to be verified by the algorithm. If a seed produces more anchors than the soft anchor cap, a subset of anchors is selected that is equal in size to the soft anchor cap. The soft and hard anchor caps can be combined, with the hard anchor cap typically being a much larger value than the soft anchor cap. A simple way of selecting the anchors to be verified for the soft anchor cap is random sampling. The following section introduces alternative ways of subsampling anchors that exploit the structure of the bidirectional FM-Index search and may lead to a better running times or more accurate results. Exploring these possibilities could be considered particularly interesting in the context of this work, because most widely used long-read mappers rely on hash table indices and therefore are not able to make use of this structure.

3.4.3 Improving Anchor Selection

The API of the FM-Index implementation used in this work [44] provides more flexible functionality than just the `count` and `locate` operations. It allows executing a user-defined callback function whenever a leaf of the search tree is reached during the search procedure. This callback function has access to the precise error configuration that the current leaf represents, the number of errors in the configuration, and a so-called *cursor*

[6]These numbers are slightly inflated, because the *locally-suboptimal* anchors from the previous section were not filtered out.

to the index. The cursor can be used immediately or at a later point to count and locate the anchors found at the current leaf. Subpanel I of Figure 16 shows an example of these cursors next to a search tree.

Additionally, the API allows to prematurely terminate the search procedure, if a given limit N on the number of anchors found is exceeded. As a simple optimization, N can always be set to the hard anchor cap plus 1. The skipped anchors are of no interest to the algorithm. They would not be verified anyway, because the hard anchor cap was exceeded.

One approach to the subsampling of anchors for the soft anchor cap is to set N to the soft anchor cap. This selects the anchors that are reported first by the implementation for the verification. This approach is likely the fastest, because it allows the search procedure to be terminated as early as possible. It is not compatible with setting a hard anchor cap, because the search is stopped before the total number of anchors is determined.

An alternative is to fully execute the search, unless the hard anchor cap is exceeded. During the search, the cursor and number of errors of each leaf are collected and stored in a list for later use. After the search has terminated, the algorithm selects anchors from the collected cursors using different strategies. Firstly, the cursors could be sorted in ascending order by the number of errors of the leaf they represent. Then, all anchors of the cursors from the front to the back of the sorted list would be located, until the number of anchors reaches the soft anchor cap or all cursors are completely exhausted. This strategy may be beneficial, because anchors with fewer errors might have a higher chance of leading to a successful verification.

On the other hand, it may happen that a seed matches many times with a low number of errors in a repetitive region and uniquely elsewhere, with a higher number of errors or with a different error configuration. It may therefore be desirable for the algorithm to select anchors from as many different cursors as possible. While repeatedly iterating through the list of sorted cursors, the algorithm could locate a single occurrence from each cursor in a round-robin fashion, rather than fully exhausting each cursor, one by one. Another way of trying to select anchors that do not fall into repetitive regions would be to sort the list of cursors by the number of anchors they contain, instead of the number of errors of the leaf they represent. The different ways of sorting the list of cursors and selecting anchors from it are visualized in the subpanels II and III of Figure 16.

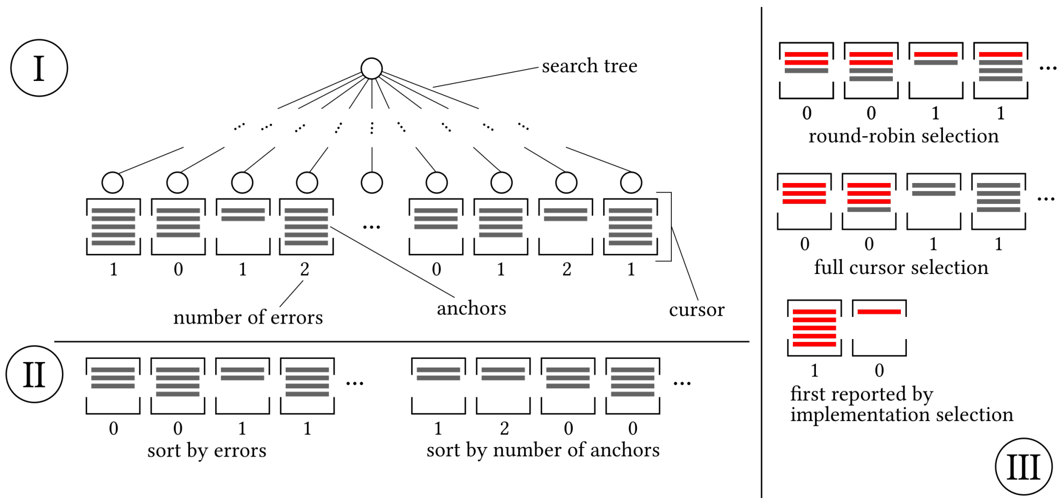

Figure 16: A visualization of how the cursors obtained from the FM-Index implementation are used to select anchors for the soft anchor cap. Subpanel I shows how leaves of the FM-Index search tree are associated with anchors. It should be noted that not all leaves of the search tree are visited in every search and therefore a cursor may not be obtained for all leaves. Subpanel II shows how the list of cursors can be sorted according to the number of errors or the number of contained anchors of the cursors. In subpanel III, different ways of selecting 6 anchors from the cursor list are depicted. In the case of the round-robin and full cursor selection procedures, the cursor list was sorted beforehand. When using the strategy of choosing the anchors that are first reported by the implementation, no unselected anchors or additional cursors exist because the search is terminated prematurely.

3.4.4 Subsampling Seeds

The prototype long-read mapper of this work implements a more approximate mode, in which not all of the pigeonhole seeds are searched. This is independent of the hard and soft anchor caps of the individual seeds. Subsampling the seeds completely breaks the exactness property of the PEX algorithm and diverges from the original goal of this work. However, it may be interesting to investigate how much running time can be saved by subsampling seeds and how many read occurrences are lost, depending on the subsampling ratio.

The subsampling is implemented by introducing a new parameter, the *seed subsampling step size*, which has to be a positive integer. The procedure is applied to the list of seeds of a single read collected from the PEX tree leaves. The first seed of a read is always searched. The index of the next seed to be searched is determined by adding the seed subsampling step size to the index of the current seed. Seeds with indices that are skipped in this procedure are not searched. For example, a seed subsampling step size of 1 does not subsample at all and selects all of the seeds. A seed subsampling step size of 2 would select every other seed.

4 Results

4.1 The Long-read Mapper Prototype Implementation

The source code of the prototype long-read mapper that implements all of the ideas proposed in work can be found in the following repository:

https://github.com/feldroop/floxer/commit/4e33b5c81332236a6accb242ec83db6245dfc0ab

The name of the tool is `floxer`, which stands for **f**m-index **lo**ngread pe**x**-based read mapp**er**. The most important libraries used in the implementation are the `fmindex-collection` library [44] for the bidirectional FM-Index search with search schemes and the `SeqAn3` library [45] for the verification/alignment implementation and IO operations. The program is multi-threaded and produces files in the SAM format as output. It allows an FM-Index to be preconstructed and stored on disk for reuse in later invocations of the program, possibly with different reads as input. In all of the benchmarks in this chapter, a pre-built index is used, because the index creation is outside the scope of this work. The index uses an alphabet consisting of the 4 DNA bases, the sentinel and the meta-character 'N', which represents an unknown base. It stores every 4-th entry of the suffix array, which can be considered a trade-off for fast `locate` running times against a larger index size of 11 GiB for the human reference genome.

`floxer` implements two different output modes. In the *reduced* output mode, the program only reports the position and score for each read occurrence. In the *full* output mode, the detailed alignment is additionally included. Downstream applications of read mappers often need the alignment data and would need to use the full output mode. However, the reduced output mode is used in most of the benchmarks of this work, because the traceback step of the verification algorithm used to obtain the detailed alignments turned out to be a significant running time bottleneck of the whole program. Most of the running time in this step is spent allocating the memory of the dynamic programming matrix for the traceback. As the performance of memory allocation in a multi-threaded context can vary a lot between different executions of the same code, this bottleneck made it impossible to examine the performance effects of other parameters. The best solution would be to use another existing more memory-efficient, and therefore also faster alignment/verification algorithm. However, testing and integrating such an implementation into a prototype read mapper is a complex task in itself and beyond the scope of this work.

4.2 Benchmark Setup

`floxer` supports a wide variety of parameters that allow evaluating the proposed algorithmic ideas of this work. Most of the individual benchmarks in this chapter focus on one or a few of these parameters. To avoid listing them all for every single benchmark, each parameter is set to a default value unless otherwise noted. An overview of all of the parameters and default values can be found in Table 1 and Table 2. The code for the benchmarks can be found in the following repository:

https://github.com/feldroop/msc-thesis-benchmark/commit/1fd02ec43fb6c95c12a08a92af29bbbafe4dd271

After giving an overview of the input data sets used in the analysis, the following sections of this chapter evaluate algorithmic ideas proposed in the Methods chapter using a number of different benchmarks. First, `floxer` is tested on a simulated data set to examine whether adjusting PEX trees to contain leaves with more than 0 errors provides the theoretically expected benefits. Next, the program is executed on a real, biological data set to observe how it performs differently when run on such data. The effectiveness of the bottom-up PEX tree construction of Section 3.2, the avoidance of repetitive verifications of Section 3.3 and the reduction of anchors of Section 3.4 is validated. Finally, `floxer` is compared to the state-of-the-art and widely used read mapper `minimap2` [19], [20]. Version 2.28 of `minimap2` was used in this work [46].

All of the benchmarks were run on a Linux compute server with a dual-socket system consisting of two AMD EPYC 9454 48-core processors and 1 TiB of main memory. `floxer` was compiled using the 12.2.0 version of the `g++` compiler. The `RelWithDebInfo` build type of the CMake build system was used. `minimap2` was compiled using the makefile included in the source code of the tool. For each invocation of `floxer` or `minimap2`, the programs were instructed to use 32 threads.

Parameter Name	**Default Value**	**Explanation**
output mode	`reduced`	Whether to use the `full` or `reduced` output mode.
read errors (k) / error rate	8%	The maximum number of errors (including indels) that a read occurrence is allowed to have. This can be supplied as a fixed value (read errors) or as a rate (read error rate). The error rate is converted to a fixed value by multiplying it with the read length.
seed errors (s)	1	The number of errors in the PEX tree leaves. The seeds are searched with this number of allowed errors.
hard anchor cap	∞	The hard anchor cap of Section 3.4.2.
soft anchor cap	50	The soft anchor cap of Section 3.4.2.
pex tree algorithm	`bottom_up`	The PEX tree construction algorithm to use. When set to `bottom_up`, the novel algorithm of Section 3.2 is used. When set to `top_down`, the top-down approach of Section 3.1 is used. The top-down approach with 0 seed errors is equivalent to the usage of the traditional PEX algorithm.

Table 1: The first part of a detailed overview of all of the relevant algorithmic parameters of `floxer`. Most input/output related parameters and parameters that influence the parallel execution are omitted.

Parameter Name	Default Value	Explanation
hierarchical verification	`on`	Whether to use the hierarchical verification. If this is set to `off`, every anchor immediately triggers a verification of the whole read.
interval optimization	`on`	Whether to use the optimization that stores already verified intervals from Section 3.3.
extra verification ratio (r)	0.1	The parameter from Section 3.3 that controls how much extra space is added to each side of the verification interval to make the interval optimization more effective.
remove locally-suboptimal	`on`	Whether to remove locally-suboptimal anchors as described in Section 3.4.1.
cursor list ordering	`count_first`	How to order the list of cursors when selecting anchors as described in Section 3.4.3. `count_first` sorts the list in ascending order of the number of anchors contained in the cursors. In the case of ties, the cursor with the lower number of errors is first in the list. The value `errors_first` uses the number of errors of the cursor first and the number of anchors in case of ties. The value `none` does not sort the list at all and keeps the cursors in the order as the are reported by the FM-Index implementation.
anchor selection procedure	`round_robin`	The way of choosing anchors from the sorted list according to Section 3.4.3. `round_robin` selects one anchor per cursor in a round-robin way, `full_cursor` exhausts each cursor before moving to the next and `first_reported` chooses the ones that were reported first by the FM-Index (`first_reported` should be used with cursor list ordering set to `none`).
seed subsampling step size	1	This parameter influences the subsampling of seeds from Section 3.4.4. If it is set to 1, every seed is searched. If it is set to 2, every second seed is searched, and so on.

Table 2: The second part of a detailed overview of all of the relevant algorithmic parameters of `floxer`.

4.3 Input Data Sets

The program is evaluated on two different data sets. Firstly, the reference genome of the *simulated data set* was created by repeatedly drawing the four DNA bases from a uniform distribution and appending the drawn base to a string representing a chromosome. 20 chromosomes, each 100 million bp long, were created in this way. The reference genome of the data set therefore has a total size of 1.9 GiB. The 100,000 reads of length 10,000 of the simulated data set were generated by selecting a random starting position in a random chromosome and repeatedly applying insertion, deletion and replacement operations at random positions in the read. When generating reads using this approach, it is possible for insertion and deletion operations to cancel each other out. It follows that the desired error rate of 8% is only an upper bound. However, such cancellations are unlikely to occur frequently and can be considered negligible. The uncompressed `.fastq` file containing the reads of the simulated data is 1.9 GiB in size.

For the *real data set*, the hg38 assembly of the human genome [47] was downloaded using the NCBI datasets [48] tool and used as the reference genome. The reads of the data set were generated from a whole human genome by the Whole Human Genome Sequencing Project [11] using the Oxford Nanopore MinION platform. The rel7 release of the data set was used [49]. The compressed `.fastq.gz` file containing all of the reads is 109 GiB in size. Running the not fully optimized read mapper prototype such a large data set would be infeasible and wasteful of computational resources. For the analysis in this work, only 0.1% of the reads from the full data set were randomly selected. Additionally, reads longer than 100,000 base pairs were removed from the data set to ensure that `floxer` could be run on the data set even in the more memory intensive full output mode. `minimap2` is be able to handle longer reads, but receives the exact same reads as input in the benchmarks. The resulting uncompressed `.fastq` file is 284 MiB in size. Figure 17 shows the distribution of read lengths in the real data set. The majority of the 15,513 reads have a length below 20,000 bps and the average read length is 8,394.

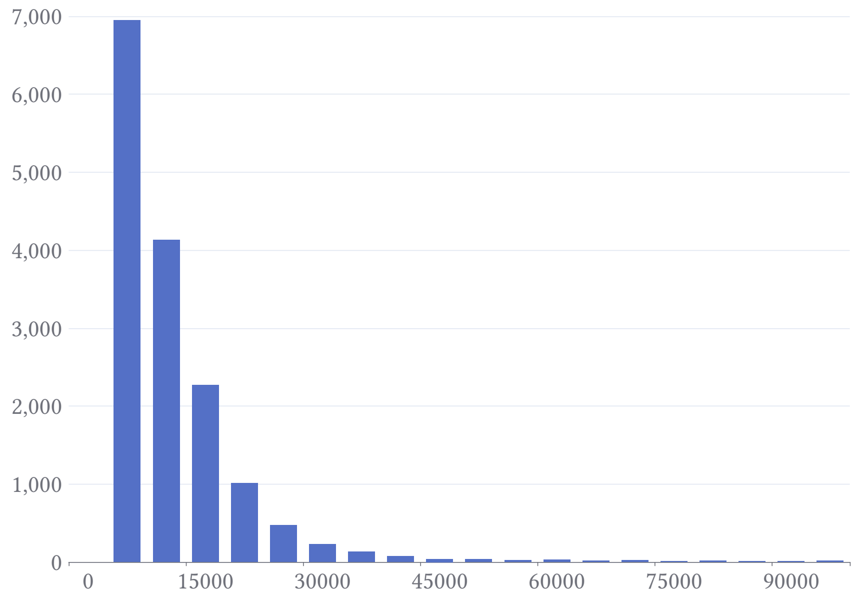

Figure 17: A histogram of the read lengths of the real data set. The average read length is 8,394 bps. Out of the total 15,513 reads, only a few hundred are above 50,000 bps in length.

4.4 General Evaluation on Simulated Data

To compare the performance of different runs of `floxer` with varying parameter settings, several metrics are used. Firstly, the quality of the generated output is assessed by the number of reads that were mapped to the correct location. In the real data set, determining the correct mapping location is not possible. However, in the simulated data set, it is possible to test whether a read occurrence was found at the position where the read was extracted from the simulated reference genome. It is expected that `floxer` maps all reads to the correct location if the hard and soft anchor caps are both set to ∞. Secondly, the computational performance of the program is analyzed by comparing the running time and memory usage.

All of the benchmarks in this section have been run on the simulated data set.

4.4.1 Comparison of Different Numbers of Seed Errors

In the first benchmark, `floxer` was run on the simulated data set with the seed errors parameter set to the values 0, 1, 2 and 3. The other parameters were set to their default values. Although this means that a soft anchor cap of 50 was used and the algorithm is therefore not guaranteed to find every read occurrence, all of the 100,000 reads were mapped correctly in all of the cases. Figure 18 shows the running time and memory usage of the different runs. The run with 0 seed errors has the longest running time, both in CPU and wall time. This likely can be explained by the large number of random, false

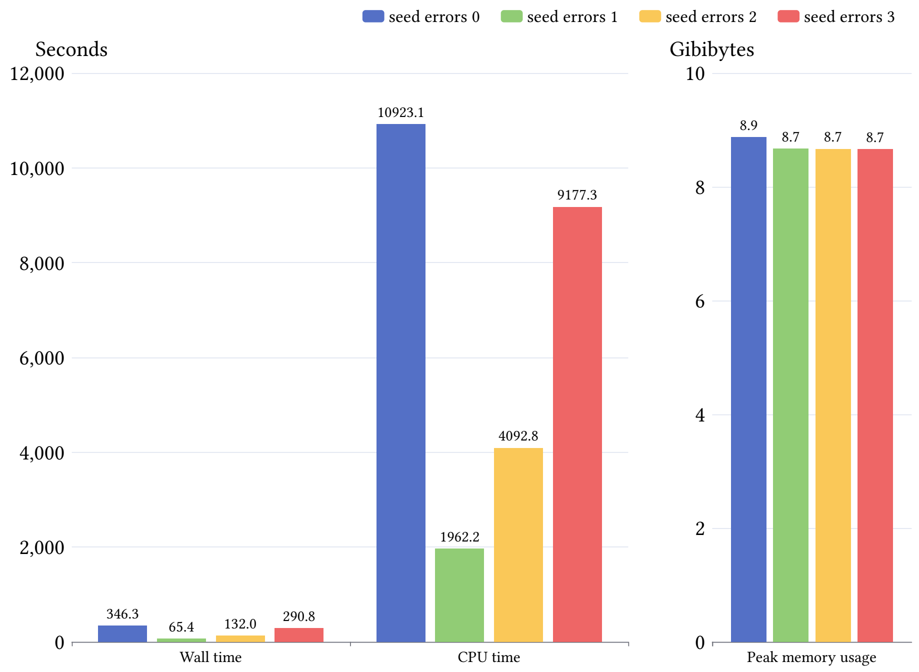

Figure 18: The computational resource metrics for the execution of `floxer` with a different number of errors in the PEX leaves (seed errors). While the memory usage does not show any significant difference, the shortest running time with one seed error is about five times faster than the longest running time with zero seed errors. The CPU times are all around 32 times higher than the corresponding wall times, which indicates a good load balancing of the parallelization in all cases.

positive anchors produced when searching very short seeds in a large genome. The run with a single seed error is the fastest. As explained in Section 3.1, having a single error in the leaves of the PEX tree likely is sufficient to avoid a large number of random, false positive anchors for references as large as the human genome. Therefore, searching with 2 or 3 errors does not provide any benefit in this case and only makes the search itself more expensive. The memory usage does not show a significant difference between the runs. Most of it is explained by the 6.4 GiB large index, which is exactly the same in all of the runs.

Figure 19 supports this interpretation of the running times. It depicts the average length of the searched seeds and the number of kept anchors, after deleting locally-suboptimal anchors and applying the soft anchor cap. Measuring the number of anchors in `floxer` is difficult and the numbers should be interpreted with caution. The selection of anchors under the soft anchor cap is done *before* the `locate` step, to avoid unnecessarily locating anchors that are thrown away immediately afterwards. However, the deletion of locally-suboptimal anchors must be done *after* the `locate` step, because it needs to know the position of the anchors on the reference. Therefore, it is impossible to tell how many of the anchors that were not selected when applying the soft anchor cap were locally-suboptimal. For this reason, the number of anchors remaining after all of the filtering steps is the focus of this analysis. Figure 19 shows how the short seeds with 0 errors produce a large number of anchors. In all other cases, approximately one anchor is found per seed, which is the desired behavior.

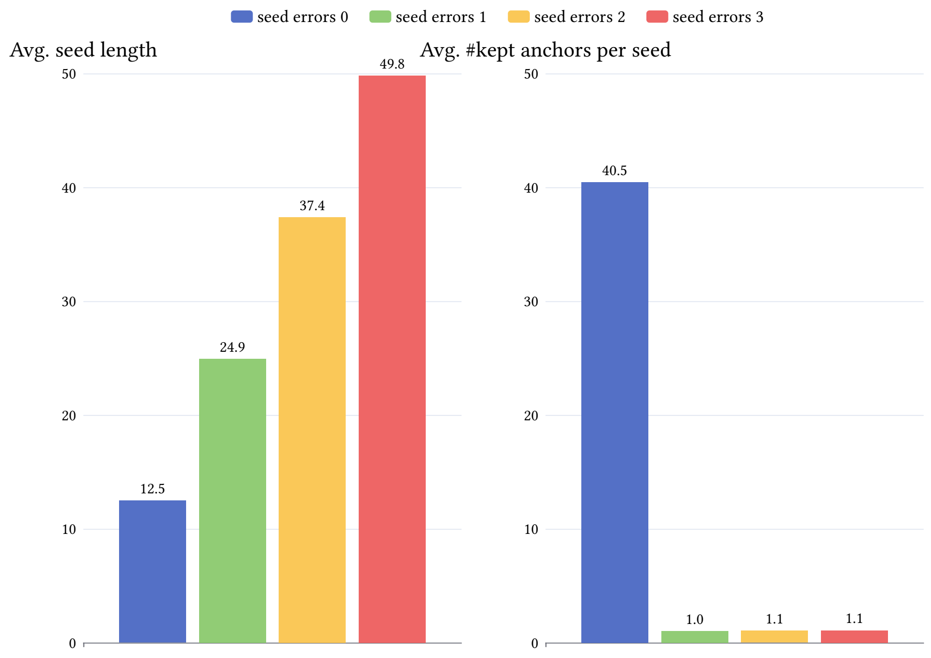

Figure 19: For different numbers of seed errors, this figure shows the average seed length and the number of anchors that are kept and used to start hierarchical verifications. The seed length simply grows linearly with the number of seed errors. The number of kept anchors is slightly below the soft anchor cap of 50 for 0 seed errors, and around 1 in the other runs.

4.4.2 Running Time Profiles

Figure 20 shows how the different values of the seed errors parameter influence the running time profiles of `floxer`. It is important to note that all of the benchmarks in this section were run in the reduced output mode. In the full output mode, the verification would be the dominant part of the running time in all cases. However, the reduced output mode allows observing how the case of 0 seed errors has a significantly different running time profile than the other cases. For the other runs, the dominant operation is the search step, which also explains the difference in the total running time between them. As expected, a larger number of seed errors makes the search more expensive. However, for 0 seed errors, the verification is by far the largest chunk. The locate step takes about 10% of the running time, while being invisible in all other cases. The larger share of other operations can also be explained by the overhead of handling and accumulating larger numbers of anchors between the steps of the algorithm. Motivated by this result, it may be interesting to explore whether the soft anchor cap can be used to further reduce the number of anchors without leading to a loss of results. It is expected that setting it to a lower value would have a significant positive impact on the running time in the case of 0 seed errors, as fewer anchors need to be located, verified and otherwise handled throughout the steps of the algorithm.

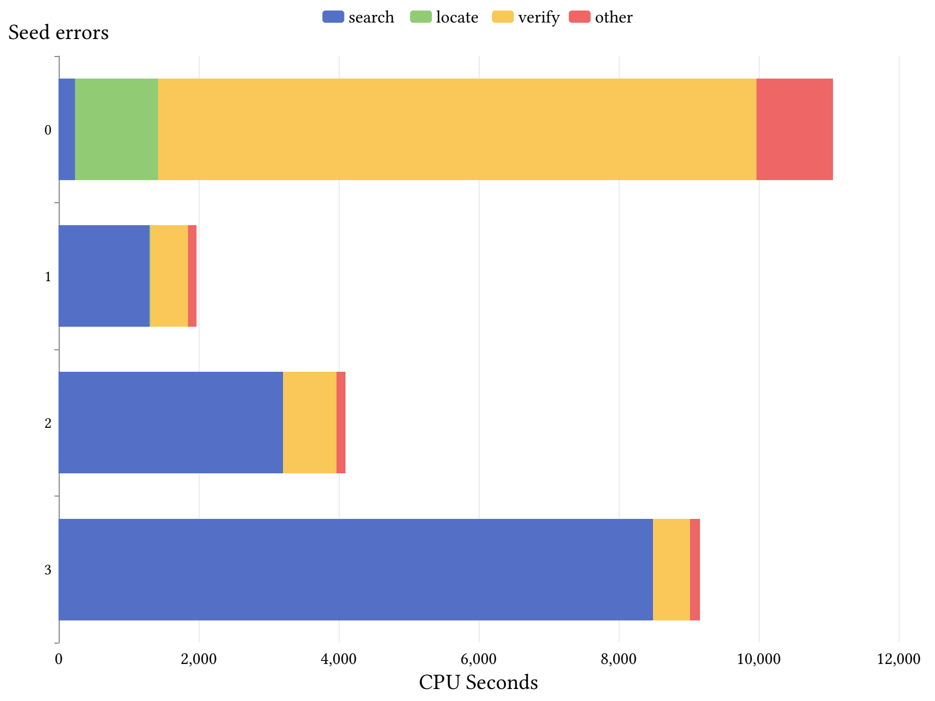

Figure 20: The running time profiles of `floxer` for runs with different numbers of seed errors. The main difference between the runs with 1, 2 and 3 seed errors is the longer search time for larger numbers of seed errors. For the run with 0 seed errors, the profile is significantly different. This is due to the large number of anchor produced by the very short seeds.

4.4.3 Comparison of Different Soft Anchor Caps

Figure 21 shows that the running time can indeed be reduced by lowering the soft anchor cap. It can be set as low as 20 without missing any read occurrences. This result could be considered surprising, because when no soft anchor cap is used, an average of about 77 anchors are kept per seed. Since the simulated reference genome does not contain repetitive regions, all but one of these anchors are expected to be random, false positives. If only 20 of the 77 anchors are sampled per seed, it could be expected that many read occurrences will be missed. However, to detect an occurrence, only a single anchor needs to be located within the occurrence. Using the parameters of this benchmark, a read of the simulated data set produces 1602 seeds on average. The pigeonhole seeding approach guarantees that in the worst case distribution of errors along the read occurrence, at least one anchor will be found. In practice, however, this worst case error distribution rarely occurs and therefore, many anchors are found within the correct read occurrence. Some of these anchors may be lost due to the soft anchor cap, without the whole occurrence being missed.

It should be noted that even with a lower soft anchor cap of 20, the program is significantly slower with 0 seed errors than with 1 or 2. Setting the soft anchor cap lower than 20 leads to the loss of results. In addition, if the anchor cap is used, there is always a

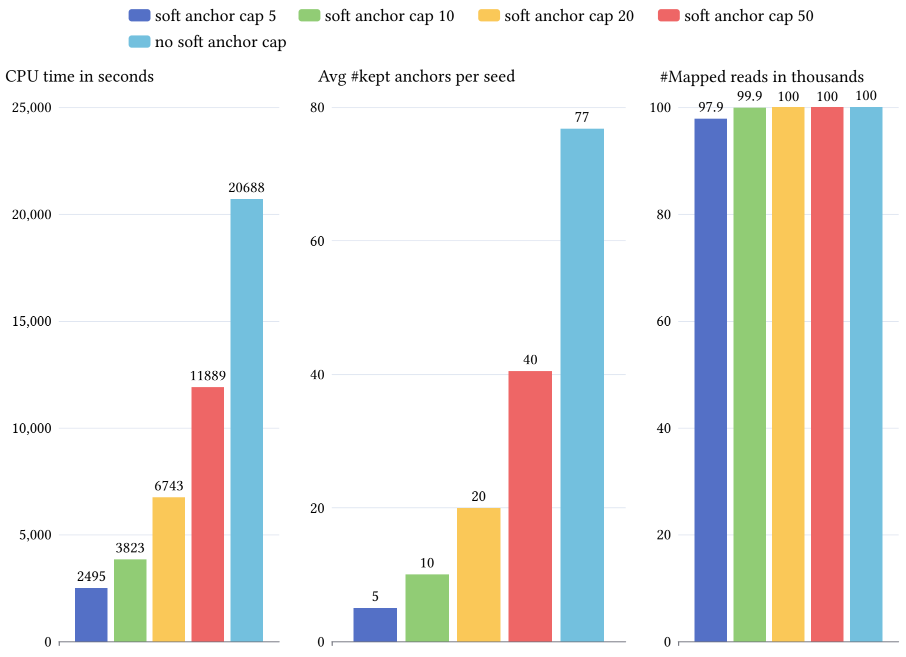

Figure 21: A comparison of several metrics for runs of `floxer` with 0 seed errors and different values of the soft anchor cap. The leftmost chart shows that lowering the soft anchor cap significantly reduces the running time. This can be explained by the middle chart, which shows the average number of kept anchors per seed. A higher soft anchor cap leads to a lower number of kept anchors, which in turn leads to less running time spent in the locate and verification steps of the algorithm. When no anchor cap was used, 77 anchors were produced per seed. The chart on the right shows that only the soft anchor caps of 5 and 10 lead to the loss of results. In the other cases, all 100,000 simulated reads were mapped correctly.

chance of missing read occurrences. With 1 or more seed errors, there is no need to use an anchor cap, because almost no random, false positive anchor are found that hurt the performance.

For the simulated data set, the state-of-the-art read mapper `minimap2` using the parameter preset intended for noisy Oxford Nanopore long-reads was 2.7 times faster than `floxer` in reduced output mode and 13.6 times faster than `floxer` in full output mode. Although `minimap2` makes extensive use of heuristics, it only missed a single read occurrence out of the total 100,000. A more detailed comparison between `minimap2` and `floxer` on the real data set can be found in Section 4.9.

4.5 General Evaluation on Biological Data

For the real data set, it is not possible to know the correct location of a read in the reference genome. Therefore, it is not possible to assess the quality of the resulting output by determining whether the read has been mapped to the location where it originated. One possibility is to compare the results to a state-of-the-art read mapper such as `minimap2`, which is be done in Section 4.9. In this section, the number of reads for which at least one occurrence was found is instead used to compare runs of `floxer` with different algorithmic parameters. This number is also referred to as the number of mapped reads. The use of this metric can be considered a viable approach, because many downstream applications are interested in the best or a single good alignment per read. In addition, every read that is generated by the sequencing machine has a single location in the genome of the sample where it originated.

All of the benchmarks in this section have been run on the real data set.

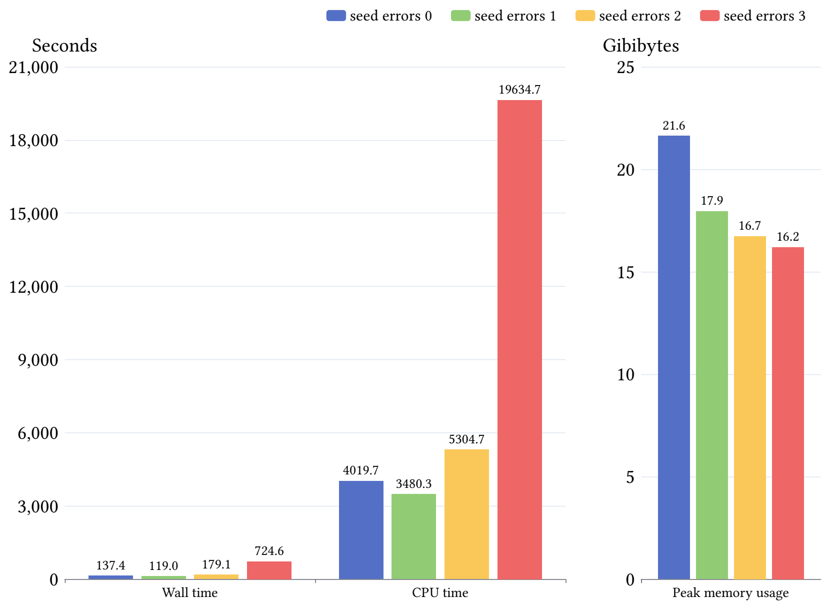

Figure 22: The resource usage metrics when running `floxer` on the real data set with different values of the seed errors parameter. The run with 1 seed error is the fastest. However, the difference to the run with 0 seed errors is small. The run with 3 seed errors takes by far the longest time. In contrast, the memory usage of the run with 0 seed errors is the highest of all of the runs.

4.5.1 Comparison of Different Numbers of Seed Errors

Figure 22 shows the metrics of computational resource usage when running `floxer` on the real data set with 0, 1, 2 and 3 seed errors. The wall times are roughly equivalent to the CPU times divided by 32, which again indicates a good load balancing and efficiency of the parallelization. In contrast to the runs on the simulated data set, the run with 0 seed errors is only slightly slower than the run with 1 seed error. However, the memory usage of the run with 0 seed error is now higher than the memory usage of the other runs. This may be due to the necessity of storing a large number of anchors and their associated bookkeeping data structures. The run with 3 seed errors has by far the longest running time, with an even larger difference than on the simulated data set. This is likely due to the more expensive search with a higher number of seed errors.

Figure 23 shows the number of reads with at least one detected occurrence (#Mapped reads) and the number of kept anchors per seed, depending on the number of seed errors of the respective runs of `floxer`. The number of kept anchors again refers to the number of anchors remaining after applying the soft anchor cap and filtering out locally-suboptimal anchors. In general, occurrences could only be detected for around 55% of the reads in the data set. The reasons for this low number are investigated in Section 4.9. The number of mapped reads increases with the number of seed errors, with the largest jump occurring from 0 to 1 seed errors. This could be due to the large number of random, false positive anchors produced by the short seeds in the case of 0 seed errors. The anchors reduce the probability of selecting a useful anchor under the soft anchor cap.

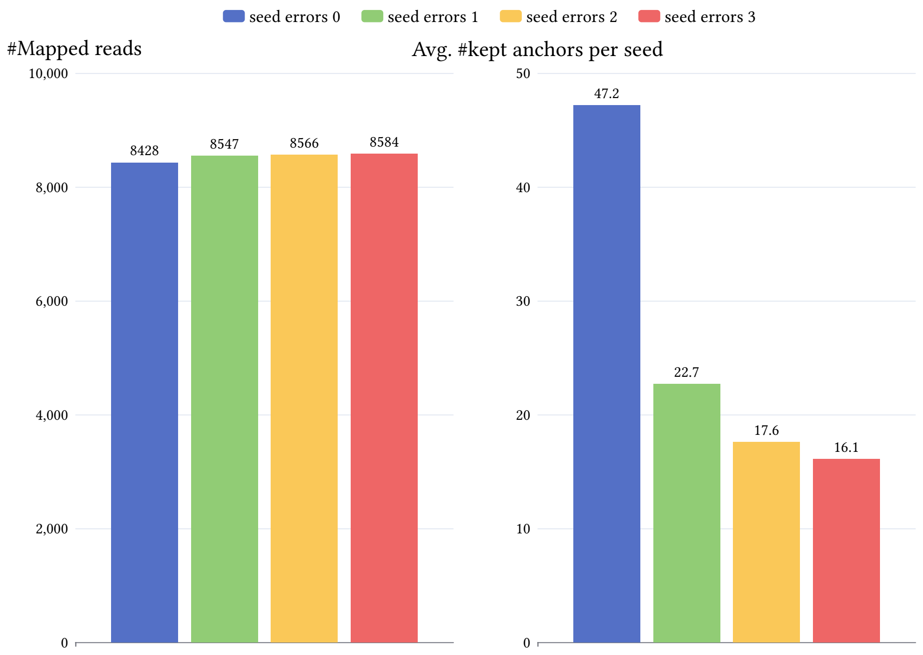

Figure 23: For different numbers of seed errors, this figure shows the number of reads with at least one detected occurrence (#Mapped reads) and the average number of kept anchors after the soft anchor cap and filtering of locally-suboptimal anchors. The number of mapped reads increases with larger numbers of seed errors, while the average number of kept anchors drops. In both cases, the largest difference is between the runs with 0 and 1 seed errors.

The difference in terms of the number of mapped reads between the other runs with 1, 2 and 3 seed errors cannot be explained in this way, because not as many random, false positive anchors are expected for these values.

The number of anchors on the right side of Figure 23 may provide an explanation for this phenomenon. The number of kept anchors is almost equal to the soft anchor cap of 50 for the run with 0 seed errors, similar to the behavior seen on the simulated data set. However, the other runs do not average around 1 kept anchor as on the simulated data set. Their number of kept anchors is around 20, with larger numbers of seed errors resulting in slightly lower numbers of kept anchors. The generally higher number of kept anchors can likely be explained by the presence of repetitive sequence in the human reference genome. The longer seeds with 2 or 3 seed errors may in some cases not match at certain repeat sites where the shorter seeds do match. Therefore, they produce fewer and more specific anchors. This could explain the lower average number of kept anchors and the slightly higher number of mapped reads. In summary, given the long running time of the run with 3 seed errors, 1 or 2 seed errors might represent the best trade-off for practical applications.

4.5.2 Comparison of Different Soft Anchor Caps

Similar to the behavior observed when running `floxer` on the simulated data set, Figure 24 shows that the soft anchor cap also has a strong impact on the running time of `floxer` on the real data set. The figure compares runs with 0 and 1 seed errors for different soft anchor caps. On the real data set, the program cannot be run without an anchor cap, therefore no runs without a soft anchor cap are included. Runs with 2 and 3 seed errors were also excluded, as they show very similar behavior to the 1 seed error runs.

For both 0 and 1 seed errors, the program runs faster with smaller soft anchor caps. This is as expected, because fewer anchors have to be located, verified and otherwise handled. However, the soft anchor cap influences the running times of the runs with 0 seed errors more strongly. This could be explained by the fact that the verification of a large number of random, false positive anchors can be avoided with smaller soft anchor caps. For soft anchor caps of 5 and 10, the runs with with 0 seed errors are even faster than the corresponding runs with 1 seed error. For larger soft anchor caps, the runs with 1 seed error are faster, with the largest difference being observed between the runs with soft anchor caps of 100.

The number of mapped reads is also influenced more strongly by the soft anchor cap for the runs with 0 seed errors. While the number of mapped reads increases by only about 40 for the runs between the 1 seed error runs with soft anchor caps of 5 and 100, it increases by about 600 for the runs with 0 seed errors. This can again be explained by

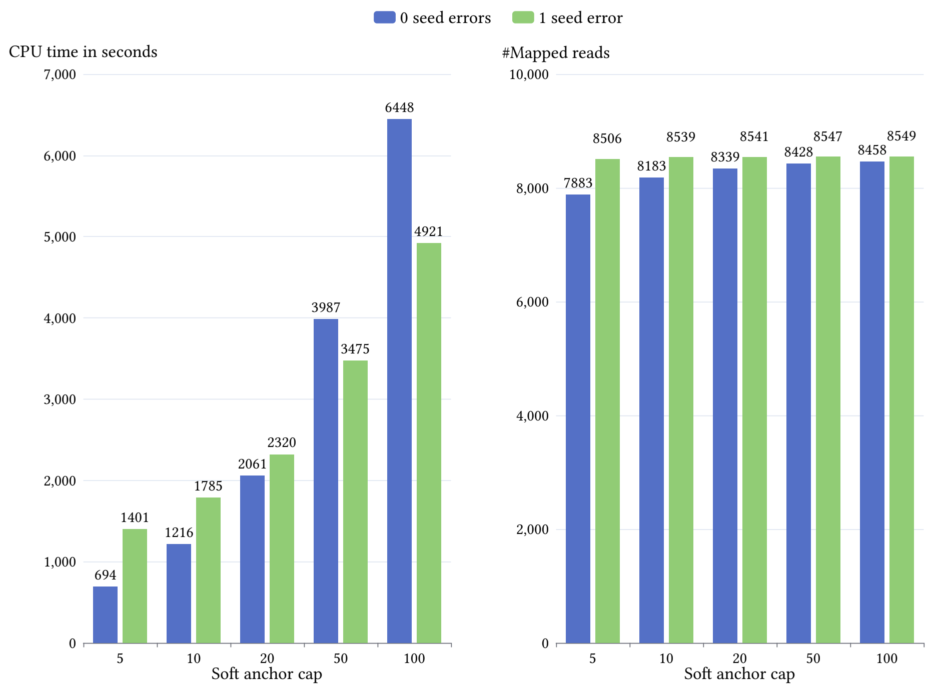

Figure 24: The CPU running time and number of reads with at least one detected occurrence for runs of `floxer` on the real data set with 0 or 1 seed errors and different soft anchor caps. For smaller soft anchor caps, the runs with 0 seed errors are faster and for larger soft anchor caps, the runs with 1 seed error are faster. The number of mapped reads increases only slightly for the 1 seed error runs with larger soft anchor caps. For the runs with 0 seed errors, more read occurrences are lost with small anchors caps.

the random, false positive anchors produced by the short seeds in the runs with 0 seed errors. These useless anchors may be selected instead of anchors that are located in an actual read occurrence. If only a few anchors per seed are selected when the soft anchor cap is set to a small value, this effect leads to the loss of read occurrences with a higher probability.

4.5.3 Running Time Profiles

To support the interpretation of why the running times of `floxer` change with different parameter sets, running time profiles of the program are analyzed in the following paragraphs. Figure 25 shows the profiles for runs with 0, 1, 2 and 3 seed errors. For 0 and 1 seed errors, the verification takes by far the largest part of the running time. The difference between the runs with 0 and 1 seed errors can be explained by the longer time needed for the locate, verification and other steps of the algorithm. All of these steps are affected by the presence of a large number of random, false positive anchors. For 2 and especially 3 seed errors, the search step starts to dominate the running time. In the cases with 1, 2 and 3 seed errors, the verification step takes more time in relation to the total execution time when run on the real data set than on the simulated data set. This could be explained by the small number of very long reads in the real data set, which make the execution of the quadratic verification algorithm expensive.

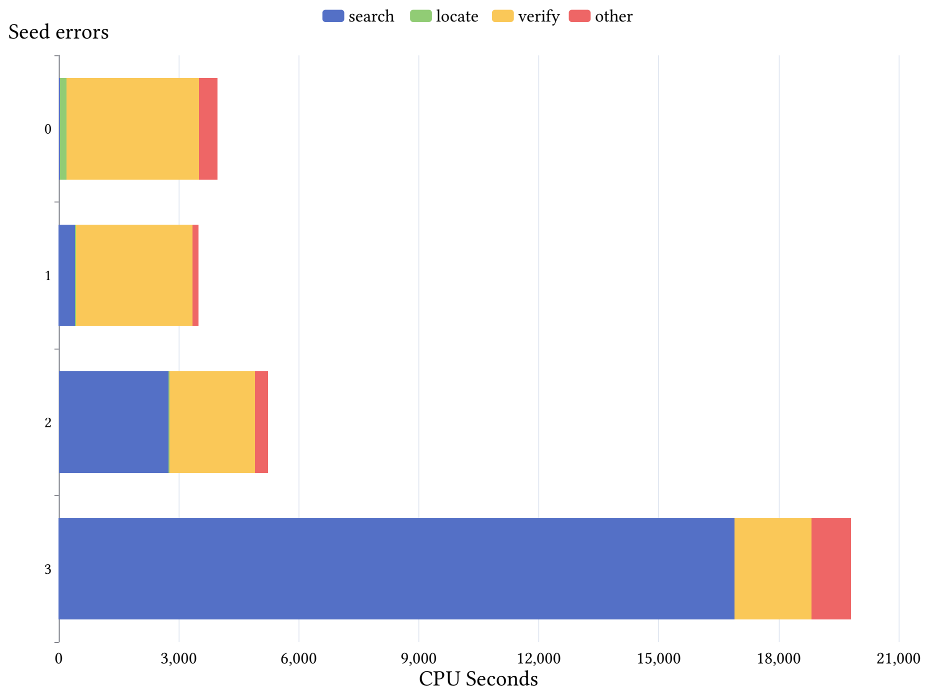

Figure 25: The running time profiles of `floxer`, executed on the real data seed with different numbers of seed errors. For smaller numbers of seed errors, the verification is the dominant part of the running time, even in the reduced output mode. For 2 and especially 3 seed errors, the search takes up most of the running time.

When the full output mode is used, the verification takes significantly more time in all cases both in absolute terms and relative to the total execution time of the program. However, the running time profiles look only slightly different, because the biggest difference between the full and reduced output mode is in the worse parallelization due to memory allocation bottlenecks. This behavior is analyzed in detail in the next section.

4.5.4 The Full Output Mode

Figure 26 shows the metrics of computational resource usage for runs of `floxer` with different seed errors using the full output mode. It should be noted that the output mode does not alter the amount of detected read occurrences. In the previous visualizations, the user and system CPU times were not displayed separately, because the system CPU time was negligible. In the full output mode, this is not the case, so the two metrics are displayed separately. The resulting numbers significantly differ from the program behavior observed in the reduced output mode. In terms of wall time, the run with 2 seed errors is now the fastest and the run with 3 seed errors is no longer by far the slowest. Instead, the run with 0 seed errors is now the slowest. For 0 seed errors, `floxer` is about 20 times faster in the reduced output mode than in the full output mode. For 3 seed errors, the difference is only threefold.

Looking at the CPU times, the wall and CPU times are not directly proportional. This could be explained by the fact that large parts of the running time of the program are

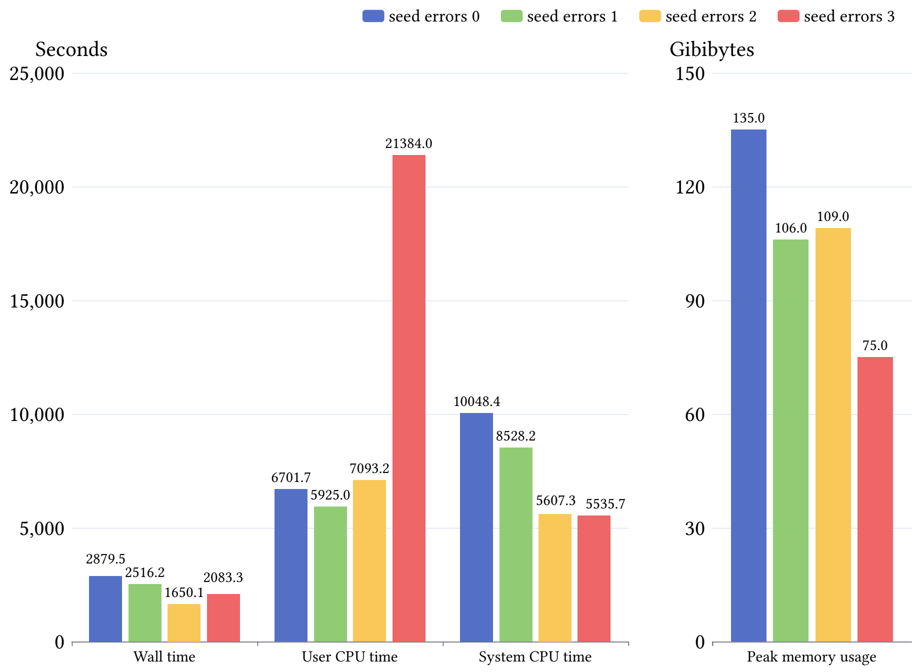

Figure 26: An overview of the computational resource metrics when running `floxer` with different numbers of seed errors in the full output mode. The run with 0 seed errors is the slowest in terms of wall time and the run with 2 seed errors is the fastest. The user CPU time looks differently. Here, the run with 3 seed errors takes by far the most time. Additionally, significant parts of the CPU time are now spent in system functions. The memory usage of the run with 0 seed errors is the highest and almost double the amount of the memory usage of the run with 3 seed errors.

spent allocating the memory of large traceback matrices. The significant amount of time spent in system functions also points to this explanation. Allocating large sizes of memory in a multi-threaded context has unpredictable and non-deterministic effects on the performance of programs. In addition, it may often require synchronization between threads, thus reducing the effectiveness of the parallelization.

The run with 3 seed errors still spends a significant part of its total running time in the search step of the algorithm, which can be parallelized well. This could explain the high CPU and low wall time compared to the runs with fewer seed errors. Additionally, the memory usage of the run with 3 seed errors could be the lowest of all runs for the same reason. If all threads spend most of their time in the verification, they all need to keep a large traceback matrix in memory at the same time. On the other hand, if a significant part of the total running time is spent searching, the probability that all threads are running verifications with large traceback matrices at the same time is small. In general, the memory consumption of all of the runs in the full output mode is about 5-6 times larger than the corresponding runs in reduced output mode.

4.5.5 Comparing Different Read Error Rates

Next, `floxer` is tested for different read error rates. This does not mean that the input data set is changed or modified. Instead, the read error rate refers to the parameter of `floxer` that is used to derive the number of allowed errors for the PEX algorithm, as described in Table 1. It can be understood as an upper bound on the expected error rates of the input reads. So far, all benchmarks have been run using the default error rate of 8%. The expectation is that the program runs slower with higher error rates, because the PEX trees have more leaves, resulting in more, shorter seeds that have to be searched and in turn more anchors that have to be verified. On the other hand, the program may be able to detect slightly more read occurrences. This could stem from the fact that even though Oxford Nanopore reads typically have an error rate of at most 8%, there may exist reads of exceptionally low quality, or regions in the sample genome with particularly high genetic variance compared to the reference genome. With high error rates, it may even be possible to detect occurrences with small structural variants, even though they are outside the scope of this work.

However, Figure 27 does not show the expected results. It depicts the running time in CPU seconds and number of mapped reads for runs of `floxer` with increasing read error rates, from 5% to 15% in steps of 2%. The runs from 5% to 11% behave as expected. Both the running time and number of mapped reads increase with higher error rates. Especially for the lowest error rates, the program is only able to detect occurrences for a small number of reads. This indicates that the estimate of an error rate of 8% for the real data set was not an an underestimate.

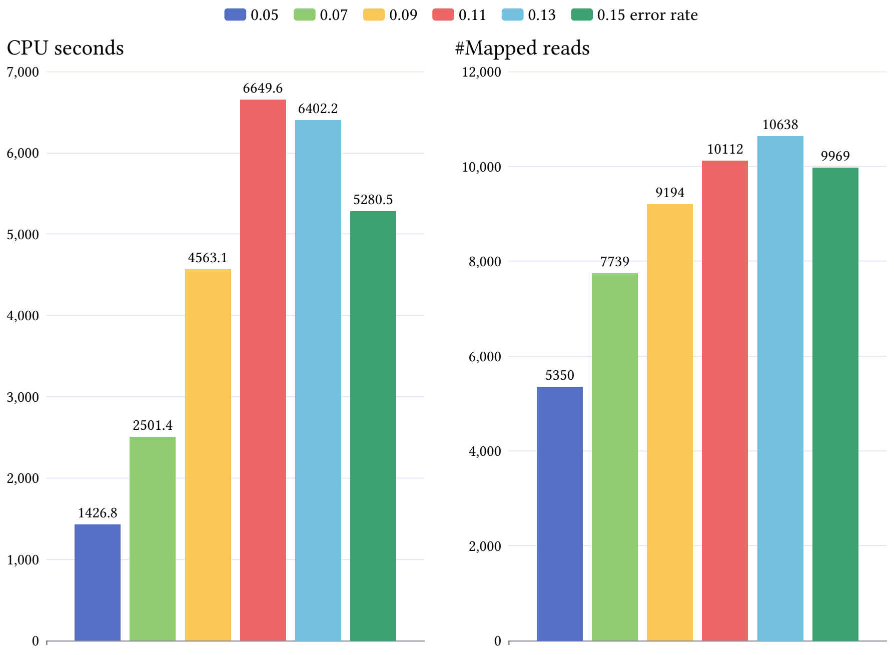

Figure 27: A comparison of the CPU time in seconds and the number of reads with at least one detected occurrence for runs of `floxer` with different read error rates. At lower error rates, both the CPU time and the number of mapped reads increase. However, both metrics reach a maximum and start to decrease at high error rates.

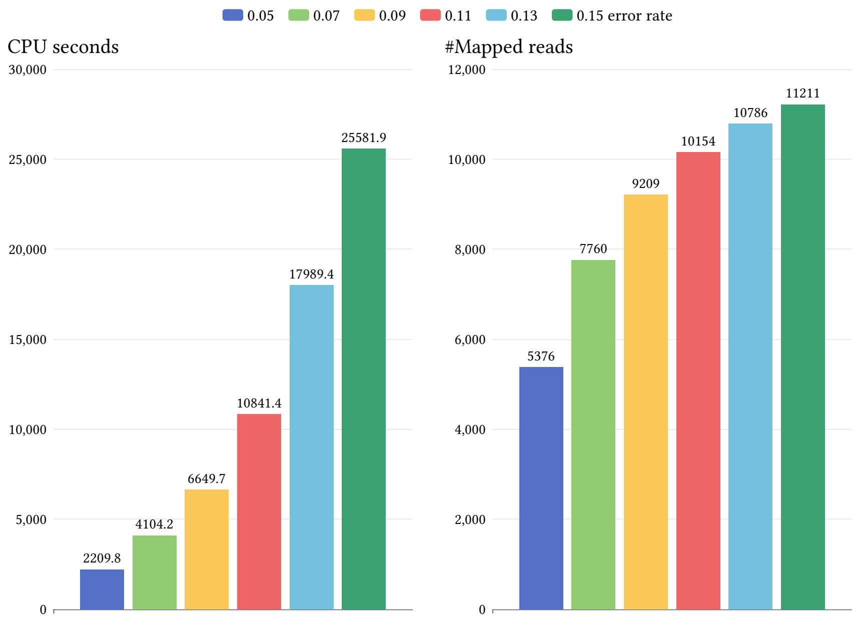

Figure 28: The CPU time and number of mapped reads for runs of `floxer` with different read error rates and 2 seed errors. Both metrics increase with higher error rates. The largest individual increases of CPU time occur between higher error rates. On the other hand, the largest individual differences in the number of mapped reads are observed between lower error rates.

The surprising part of Figure 27 is that both the CPU time and the number of mapped reads start to decrease after reaching their maximum at error rates of 11% and 13%, respectively. Figure 28 may provide an explanation of this phenomenon. It depicts the exact same metrics as Figure 27 for runs with different read error rates, but instead of the default of 1, the seed error parameter was set to 2. With this change, the results are exactly as previously expected. The running time and number of mapped reads monotonically increase with a higher error rate. It follows that the decrease in running time and number of mapped reads for the runs with 1 seed error can likely be explained by the fact that the pigeonhole seeds become shorter at higher error rates. For 1 seed error and error rates of 13% and 15%, the seeds are so short that they start to produce a large number of random, false positive anchors, reducing the probability of useful anchors being selected under the soft anchor cap. This directly explains the lower number of mapped reads. In turn, the lower running time may occur because less costly verifications of full long-reads need to be run when many anchors are false positives.

Figure 28 also shows that the read error rate has a strong impact on the running time. The running time for `floxer` with a read error rate of 5% is ten times shorter than for a run with an error rate of 15%. In addition, higher error rates allow detecting significantly more read occurrences.

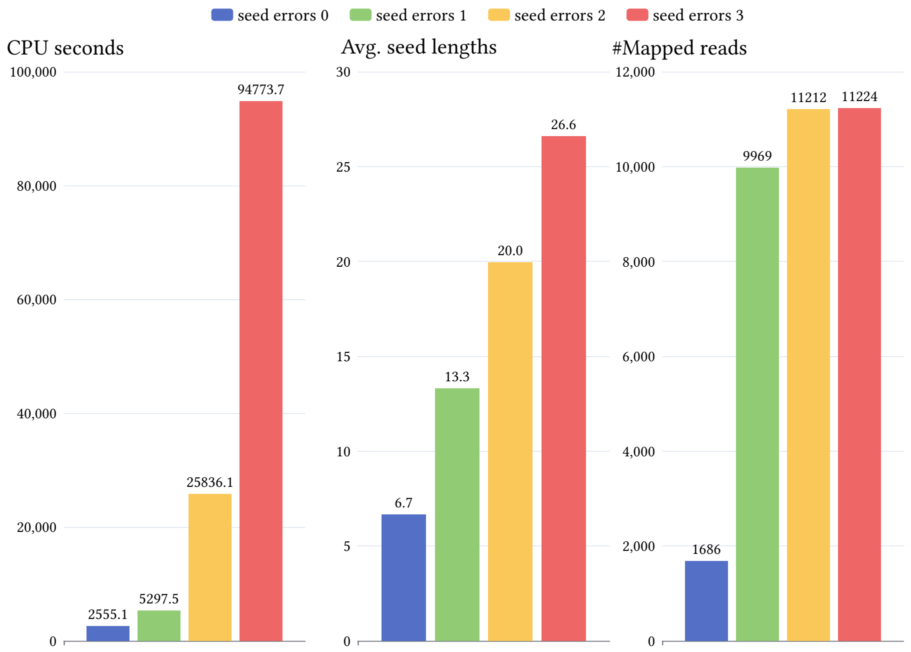

Figure 29: The CPU time in seconds, average read lengths and number of mapped reads for runs of `floxer` with different numbers of seed errors and a read error rate of 15%. The running time increases seemingly exponentially. The average seed length grows linearly, starting at a low value of 6.7. For the 0 seed error run, the number of mapped reads is very low compared to the other runs. For the run with 1 seed error, it is significantly higher. However, there still exists a notable difference to the runs with 2 and 3 seed errors, which have a similar number of mapped reads.

Figure 29 helps explan why the number of mapped reads decreased in the runs with 1 seed error at higher read error rates. It shows the CPU running times, average seed lengths and number of mapped reads for `floxer` when run with different numbers of seed errors and a read error rate of 15%. The running times increase significantly with higher numbers of seed errors. The average seed lengths show why the runs with smaller numbers of seed errors may not be able to detect as many read occurrences as the runs with 2 or 3 seed errors. Seeds of lengths around 7 and 13 are not long enough to be searched in the human genome without producing a large number of random, false positive anchors. As explained above, this leads to lower numbers of mapped reads, because a soft anchor cap has to be used when dealing with real, biological data.

4.5.6 Comparing Hierarchical and Direct Verification

The final benchmark of this section tests the effectiveness of the hierarchical verification of the PEX algorithm. Although it is expected to be an improvement over only using pigeonhole seeds and immediately verifying the entire pattern for every anchor, it could theoretically compromise running time performance. In the case where an anchor successfully leads to the detection of a complete read occurrence, walking up the PEX tree and verifying pattern substrings of increasing size was an unnecessary overhead. Due to the quadratic scaling of the running time of the verification algorithm, the root is by far the most expensive verification run, but the smaller verification could still add up and slow down the algorithm. However, as the hierarchical verification makes unsuccessful verifications of false positive anchors less expensive, it is still expected to have a

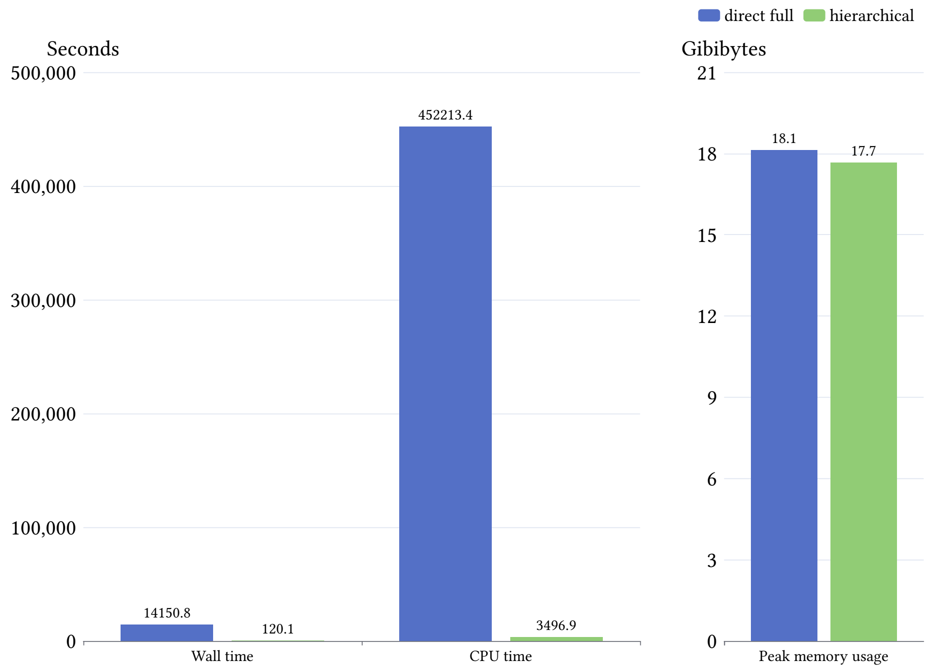

Figure 30: The running time and memory usage when running `floxer` with the different verification subroutines and otherwise using the default parameters. The wall time of the run with direct, full verification is more than 100 times the wall time of the run with hierarchical verification. The CPU times show a similar trend. The memory usage of the two runs only differ slightly.

positive impact on the running time. Figure 30 shows that the hierarchical verification significantly improves the running time performance of `floxer`. In terms of wall time, the program is more than 100 times faster when using the hierarchical verification. As expected, the number of mapped reads for the two runs of `floxer` are equivalent.

4.6 Bottom-up PEX Tree Construction Evaluation

This section compares the proposed bottom-up PEX tree construction algorithm as described in Section 3.2 to the modified top-down approach of Section 3.1. The benchmarks of previous sections used, the bottom-up algorithm. All benchmarks in this section have been run on the real data set.

The primary advantage of the bottom-up PEX tree construction algorithm is not improved performance or better results. Instead, it allows precise control of the seed error parameter. In the top-down approach, it may happen that a value for the number of seed errors is chosen, but many seeds generated from the leaves of the resulting PEX tree have a smaller number of errors. The previous sections have shown that the seed errors parameter can have a large impact on the performance and results of `floxer`. Therefore, the ability to accurately control this parameter is crucial to achieving optimal results with the program.

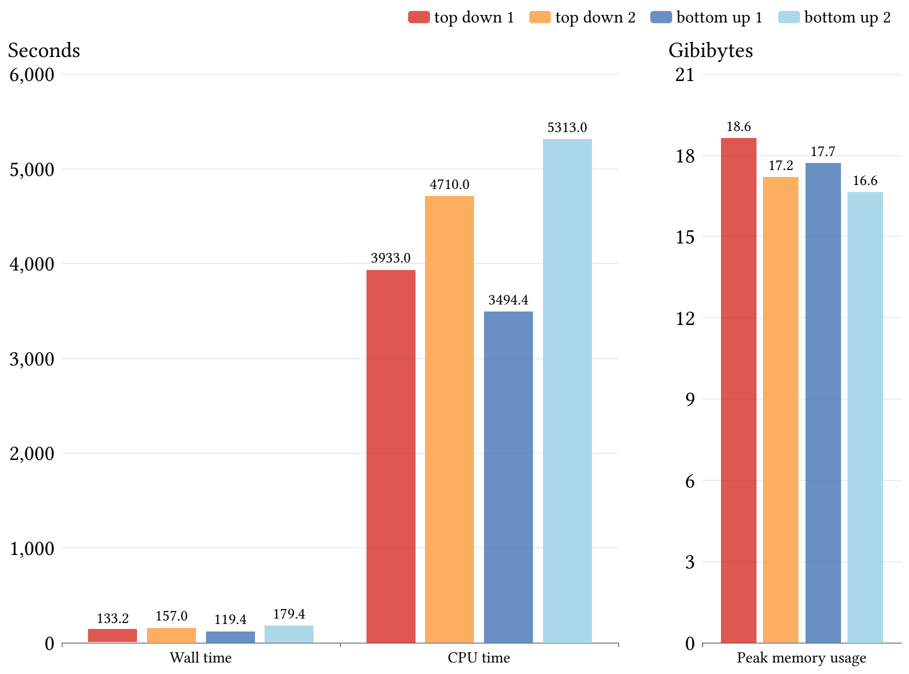

Figure 31: The metrics of computational resource usage for runs of `floxer` with different combinations of 1 or 2 seed errors and the usage of the bottom-up or top-down PEX tree construction algorithm. The bottom-up run with 1 seed error is the fastest configuration overall. On the other hand, the bottom-up run with 2 seed errors is slower than its top-down counterpart. The differences in memory usage are small, with the top-down 1 seed error run using slightly more and the bottom-up 2 seed errors run using slightly less memory than the other runs.

Figure 31 shows the metrics of computational resource usage for runs of `floxer` with different combinations of tree construction algorithms and numbers of seed errors. FOr example, the “top down 1” run refers to the usage of the top-down PEX tree construction algorithm and 1 seed error. The bottom-up run with 1 seed error is the fastest. This is likely due to the fact that seeds with 1 error are the most advantageous seeds for this configuration of parameters and input data. Seeds with 0 errors lead to many random, false positive anchors and seeds with more than 1 error are expensive to search. The top-down run with 1 seed error is slower, because the top-down PEX trees with 1 seed error contain leaves with 0 errors. The top-down run with 2 seed errors is in between the two bottom-up runs, because a mixture of seeds with 1 and 2 seed errors are searched. The differences in memory usage are small. However, the top-down run with 1 seed error has the highest memory usage, which can be explained by the large number of random, false positive anchors. The bottom-up run with 2 seed errors has the lowest memory usage, which can be explained by the fact that a significant part of the total running time is spent searching. Therefore, it is unlikely that all threads are running the more memory intensive verification step at the same time.

Figure 32 shows several metrics that explain the difference in the resource usage depicted in Figure 31. On the left, the average numbers of seed errors are compared. For the bottom-up runs, the number is equivalent to the value of the seed errors parameter. For the top-down runs, it is significantly lower, proving the hypothesis that the seed errors

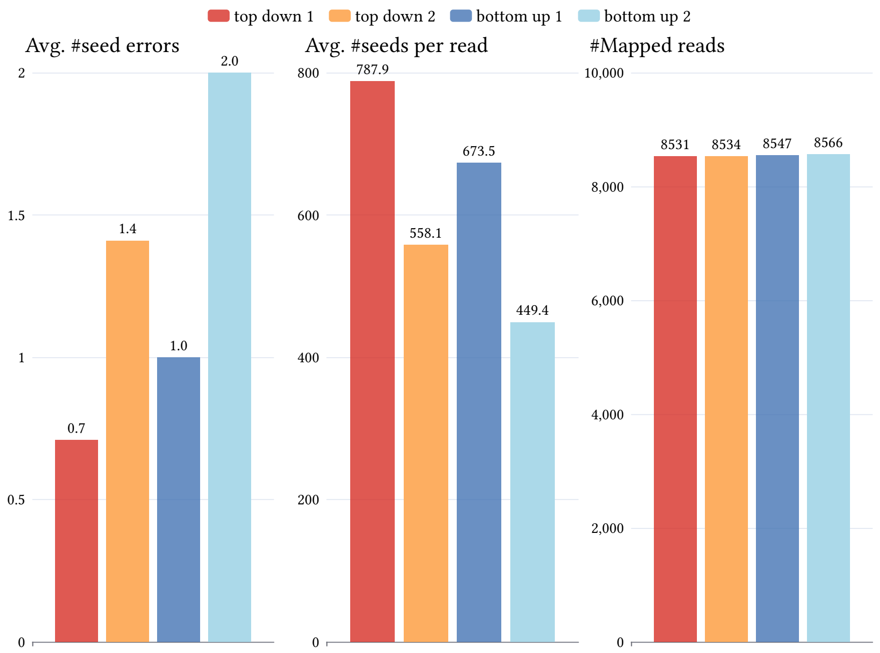

Figure 32: An overview of the average number of errors per seed, the average number of seeds per read and the number of mapped reads, for runs with different combinations of 1 or 2 seed errors and the usage of the top-down or bottom-up PEX tree construction algorithms. For the bottom-up runs, the average number of seed errors is equivalent to the value of the seed errors parameter. For the top-down runs, it is lower. A lower average number of seed errors leads to a higher number of seeds per read, as can be observed in the chart in the middle. The runs do not show significant differences in terms of the number of mapped reads.

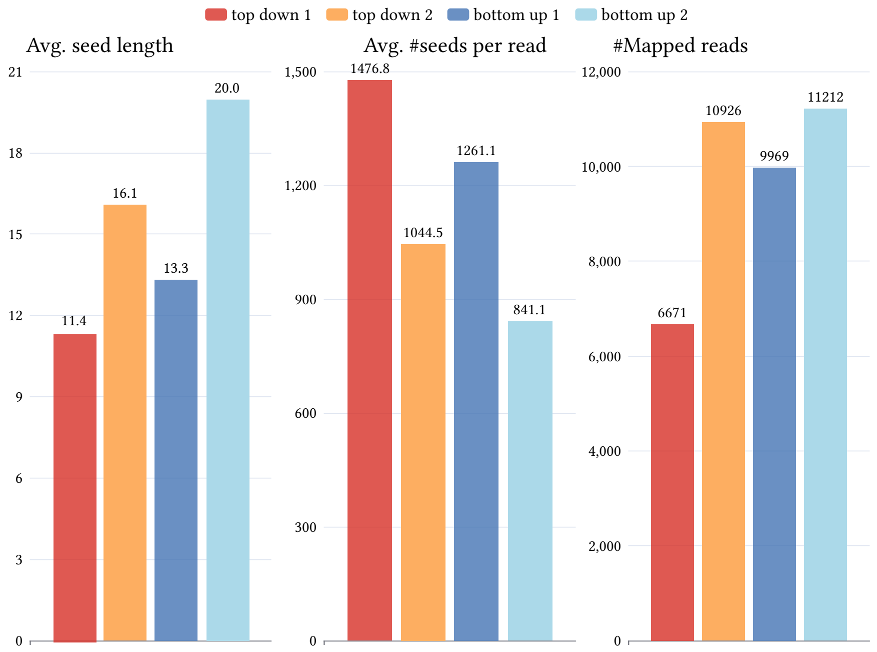

Figure 33: An overview of the average seed length, the average number of seeds per read and the number of mapped reads, for runs with a higher read error rate of 15% and different combinations of 1 or 2 seed errors and the usage of the top-down or bottom-up PEX tree construction algorithms. The chart on the left shows that the seeds for the bottom-up runs are on average longer than the seeds for the top-down runs for the same value of the seed errors parameter. The average number of seeds per read is lower for the bottom-up runs and higher for lower values of the seed errors parameter. The number of mapped reads for the top-down run with 0 seed errors is significantly lower than for the other runs. For both 0 and 1 seed errors, the bottom-up runs result in more mapped reads than the corresponding top-down runs.

parameter cannot be precisely controlled when using the top-down PEX tree construction approach. The chart in the middle shows the average number of seeds per read. These values correlate negatively with the average errors per seed. It can be observed that the bottom-up PEX tree construction approach leads to trees with fewer leaves, and in turn fewer seeds than the top-down version. The number of mapped reads, which is shown in the chart on the right of Figure 32, is similar for all of the runs. However, the top-down runs perform slightly worse than the bottom-up runs.

Figure 33 shows similar metrics as Figure 32. The differences are that it shows the average seed length instead of the average number of errors per seed and that the runs of `floxer` were performed using a higher read error rate of 15% instead of the default 8%. The average seed lengths explain why the number of mapped reads differs so strongly at such an error rate. Due to the higher average number of seed errors in the bottom-up runs compared to the top-down runs with the same value for the seed errors parameter, the seeds of the bottom-up runs are also longer. This results in fewer random, false positive anchors and in turn more detected read occurrences. The average number of seeds per read shows a similar trend as in Figure 32, only the absolute values are higher for the runs with higher error rates.

4.7 Avoiding Repetitive Verifications Evaluation

In this section, the optimization that stores previously verified intervals from Section 3.3 will be evaluated. All benchmarks in this section have been run on the real data set. It is expected that the optimization does not change the number of mapped reads and significantly improves the running time, because fewer verifications of long sequences have to be computed. The statistics recorded during the benchmarks show that `floxer` performs about 380,000 verifications of full long-reads when run on the real data set with default parameters. When the optimization that is supposed to avoid repetitive verifications is disabled, about 3,000,000 verifications are performed instead. This is roughly an eightfold increase. The number of mapped reads does not change when the optimization is disabled. Figure 34 shows that reducing the number of verification of full long-reads has a strong impact on the running time performance of `floxer` and even slightly reduces the memory usage. The run with the optimization enabled is more than 10 times faster than the run without the optimization. The fact that the running time improves by a larger factor than the number of full long-read verifications may be due to the frequent avoidance of repetitive verifications of particularly long read sequences.

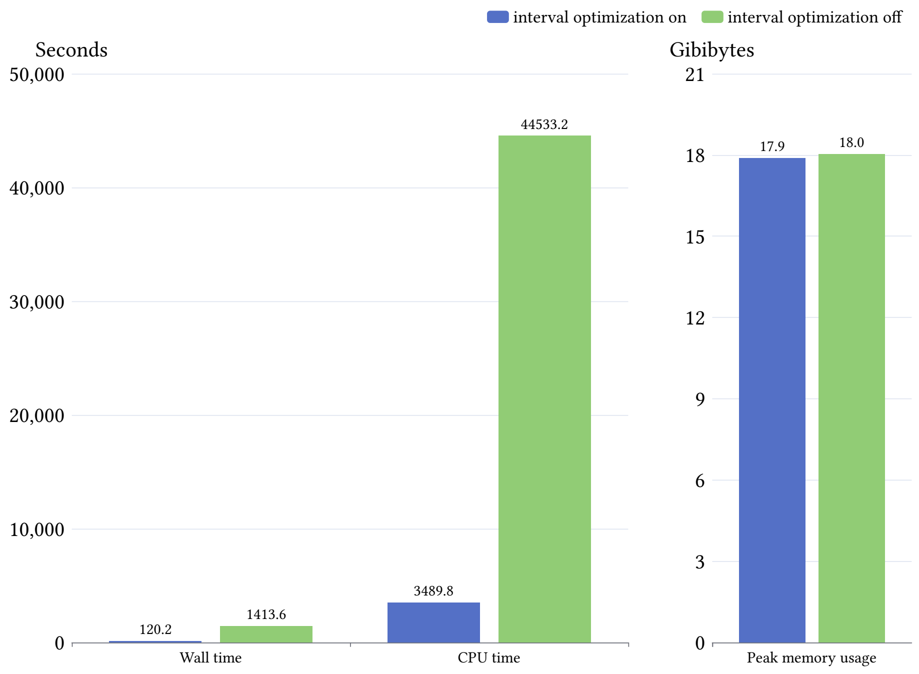

Figure 34: The metrics of computational resource usage for runs with and without the optimization that attempts to avoid repetitive verifications. It can be observed that the run with the optimization is more than 10 times faster than the run without the optimization, both in terms of wall and CPU time. The memory usage of the runs is almost the same.

The parameter r, which is called extra verification ratio in the implementation, has a significant impact on the effectiveness of the optimization. With $r = 0$, only small shifts due to indels suffice to shift the verification window by small amounts and trigger repetitive verifications. The parameter has been introduced to slightly increase the verification window such that it covers these small shifts. Figure 35 shows CPU running times for runs of `floxer` with different extra verification ratios. The same analysis was done for the reduced and for the full output mode. For the reduced output mode, larger extra verification ratios always lead to smaller running times. However, for the full output mode, the minimum running time is achieved with an extra verification ratio of 0.1. This difference could be explained by the fact that increasing the sizes of the verification interval has a stronger negative impact on the performance if the verifications are already expensive in the full output mode. For both output modes, the memory usage of the whole program was not significantly altered, even though the verifications used slightly more memory.

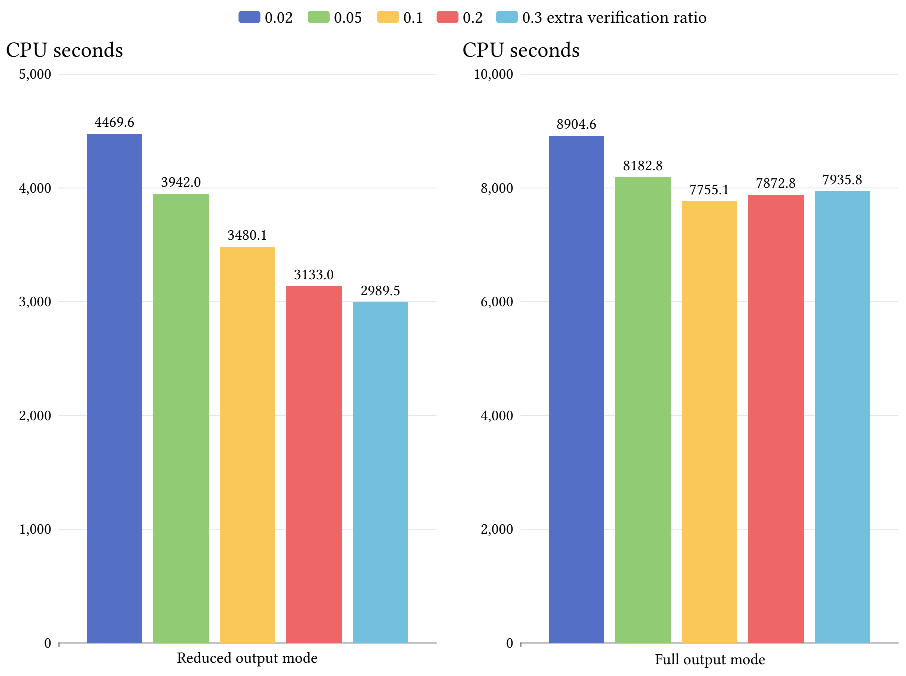

Figure 35: The CPU running times for runs of `floxer` with different values for the extra verification ratio parameter (r), for the reduced and for the full output mode. For the reduced output mode, higher extra verification ratios result in lower running times. For the full output mode, the minimum running time is reached at an extra verification ratio of 0.1.

4.8 Reducing the Number of Anchors Evaluation

All of the benchmarks in this section have been run on the real data set, if not otherwise mentioned.

4.8.1 Removing Locally-Suboptimal Anchors

This subsection evaluates the optimization of removing locally-suboptimal anchors from the set of anchors returned by the FM-Index implementation, as described in Section 3.4.1. It is expected that the optimization does not result in the loss of read occurrences and possibly improves the running time. When running `floxer` with the default parameters, 3.58 anchors are removed per seed using this procedure. The average number of kept anchors per seed is 22.69. It follows that most anchors returned by the FM-Index implementation are locally-optimal. However, Figure 36 shows that the optimization still has a positive effect on the running time and does not change the number of mapped reads. The memory usage was not significantly affected by enabling the optimization of removing locally-suboptimal anchors.

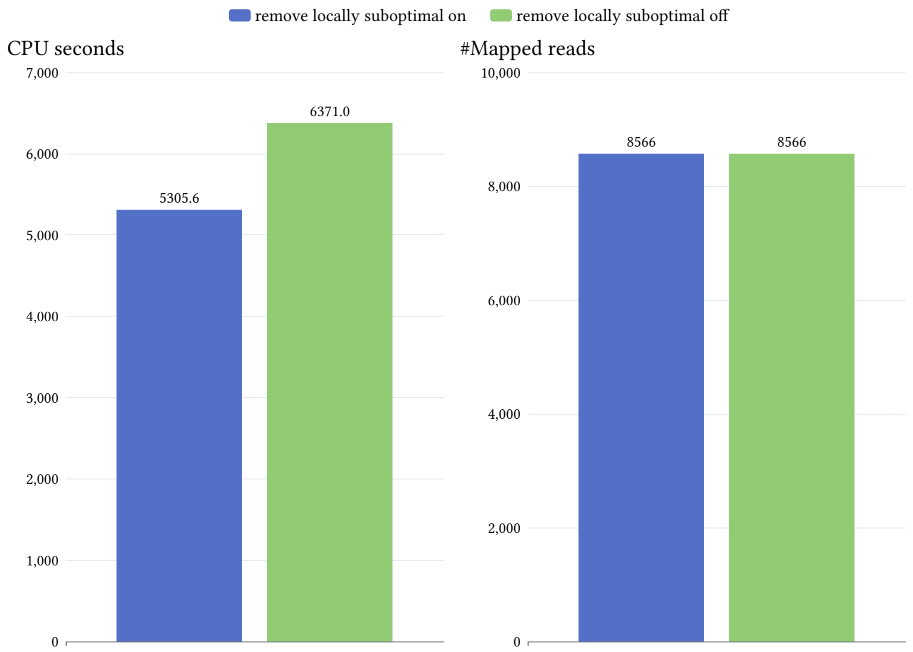

Figure 36: The CPU running time and number of mapped reads, for runs of `floxer` with and without removing locally-suboptimal anchors. While the number of mapped reads is not altered by this optimization, it improves the running time of the program by about 15%.

4.8.2 The Hard Anchor Cap

The soft anchor cap of Section 3.4.2 has already been evaluated in previous sections, because of its profound impact on the running time and number of mapped reads of `floxer`, especially for low numbers of seed errors. This subsection will therefore only focus on the hard anchor cap, which was also introduced in Section 3.4.2. Figure 37 shows the CPU running time and number of mapped reads for different hard anchor caps. It can be observed that a larger hard anchor cap leads to a higher running time and more mapped reads. The differences in the number of mapped reads are small, but not insignificant. On the other hand, the differences in the running times are substantial. The run with a hard anchor cap of 1000 is 2.8 times faster than the run without a hard anchor cap. Since the goal of this work is to be as exact as possible when mapping reads, the default of `floxer` was set to not using a hard anchor cap, even though this leads to a significant slowdown of the program. The memory usage of the program was only slightly altered between the runs with different hard anchor caps. This likely is due to the soft anchor cap being set to the same default value of 50 in all of the runs.

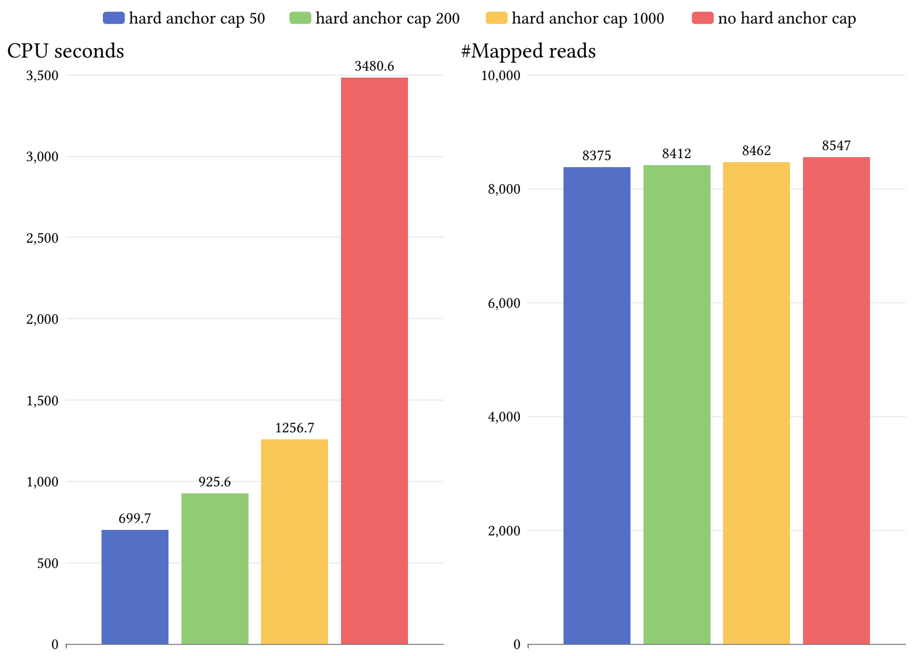

Figure 37: The CPU running time and number of mapped reads for runs of `floxer` with different hard anchor caps. Both of the metrics grow with larger hard anchor caps. The number of mapped reads grows slowly, while the differences in terms of running time are much larger. There is an especially large difference between using a hard anchor cap of 1000 and not using a hard anchor cap.

4.8.3 Anchor Selection Strategies

In this subsection, the different strategies for selecting anchors under the soft anchor cap from Section 3.4.3 will be evaluated. Each strategy consists of two parts, the ordering of the cursor list and the procedure by which anchors are selected from the list. These two concepts refer to the “cursor list ordering” and “anchor selection procedure” parameters described in Table 2. If the ordering is set to `none`, the list will not be sorted. The `first reported` selection procedure stops the FM-Index search when a number of anchors equal to the soft anchor cap is found. This is expected to be the fastest strategy. Additionally, the `count first` ordering and `round robin` selection procedures have been designed to possibly increase the number of mapped reads by selecting a more diverse set of anchors in terms of number of errors and error configuration.

However, Figure 38 shows results that do not confirm these expectations. Firstly, the number of mapped reads is not significantly influenced by the anchor selection strategy. On the other hand, the CPU running times show substantial differences. The fastest strategies are those employing the `round robin` selection procedure. The strategies that do not sort the list of cursors are the slowest. It is difficult to find an explanation for this behavior. However, the middle chart of Figure 38 shows that the faster anchor selection strategies lead to a lower number of alignments of full long-reads that are performed throughout the whole program run. This effect may be due to fewer anchors being selected that match in repetitive regions where a large number of occurrences of the same read exist. Such repetitive occurrences are not useful for most downstream applications and are not included in the output of most read mappers.

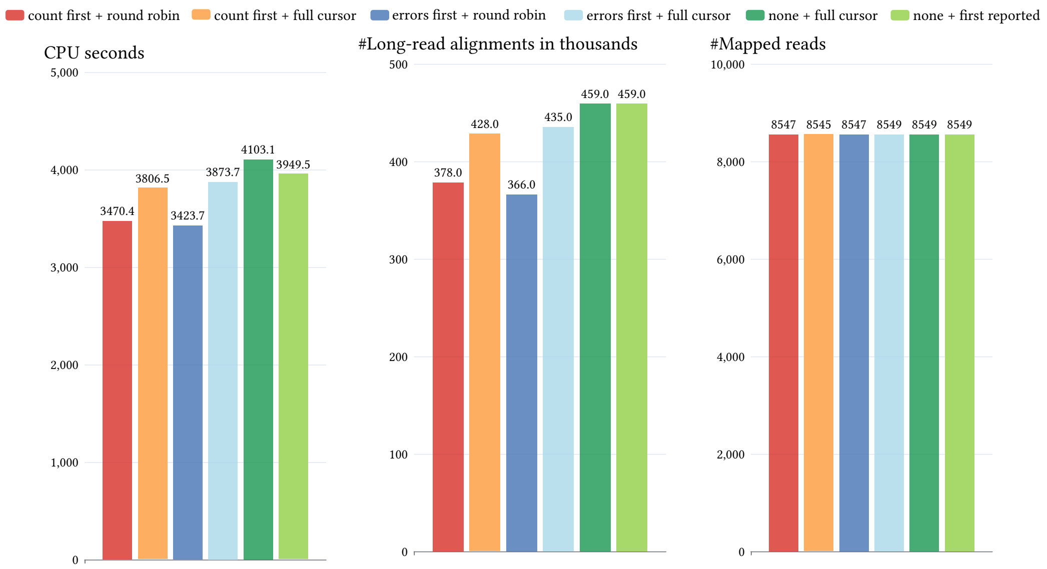

Figure 38: The CPU running time, number of alignments of full long-reads performed during the entire program run and number of mapped reads for runs of `floxer` with different anchor selection strategies. The runs are named by their cursor list ordering first and their selection procedure second. The runs with the `round robin` selection procedure have the lowest running time and number of alignments. The runs with no cursor list ordering have the highest number of alignments and are the slowest, with the `none first reported` run being slightly faster. The numbers of mapped reads are almost equal in all of the runs.

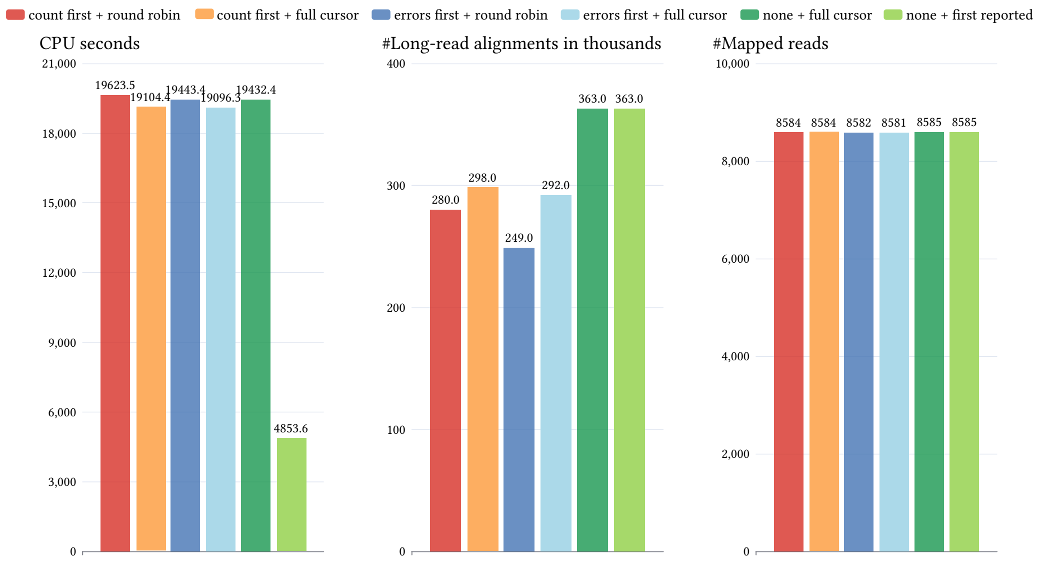

Figure 39: The CPU running time, number of alignments of full long-reads performed throughout the entire program run and number of mapped reads for runs of `floxer` with different anchor selection strategies and 3 seed errors. The runs are named by their cursor list ordering first and their selection procedure second. While the number of alignments and number of mapped reads behave similarly to the runs with 1 seed error from Figure 38, the running time of the `none first reported` run has changed to be by far the shortest.

The run with no cursor list ordering and the `first reported` selection procedure is only slightly faster than the run with no ordering and the `full cursor` selection strategy. This can be explained by the small fraction of the total running time of `floxer` that is spent in the search, when searching with the default of 1 seed error. Figure 39 shows the CPU running time, number of full long-read alignments and number of mapped reads for runs of `floxer` with different anchor selection strategies and 3 seed errors. While the number of mapped reads also does not significantly differ and the number of full long-read alignments shows a similar behavior as in the runs with 1 seed error, the running time of the run with the `first reported` selection strategy now is about 4 times faster than the running time of the other runs. This is an influential result, because it may open up the possibility of running `floxer` with 3 seed errors while achieving competitive performance to runs with lower numbers of seed errors. The memory usage of `floxer` is not significantly altered by the anchor selection strategy parameter, both for 1 and 3 seed errors.

4.8.4 Seed Subsampling

In this subsection, the seed subsampling approach of Section 3.4.4 is evaluated. It is expected that higher values for the seed subsampling step size parameter reduce the running time and may result in the loss of read occurrences. Figure 40 shows that both the memory usage and the running time are indeed reduced when using higher seed subsampling step sizes. Figure 41 shows the average number of anchors per read and number of mapped reads for runs with different seed subsampling step sizes. The average number of anchors is reduced significantly for higher numbers of seed subsampling step sizes. This is as expected, because searching fewer seeds also results in fewer anchors.

The result that could be considered interesting is that the number of mapped reads decreases only slightly with increasing seed subsampling step sizes. On the other hand, the average number of anchors per read and running time show a substantial decrease for higher seed subsampling step sizes. This may indicate that many of the seeds searched by `floxer` are redundant. However, the simple subsampling strategy employed by the program is not able to reliably distinguish between redundant and essential seeds before they are searched. Additionally, the parallelization of the program becomes less efficient for higher seed subsampling step sizes. While the CPU time of the run with a seed subsampling step size of 4 is more than twice the CPU time of the run with the parameter set to 16, the wall times of the two runs only differ by a factor of 1.37. Further investigation is required to determine what inefficiencies in the implementation are causing this behavior.

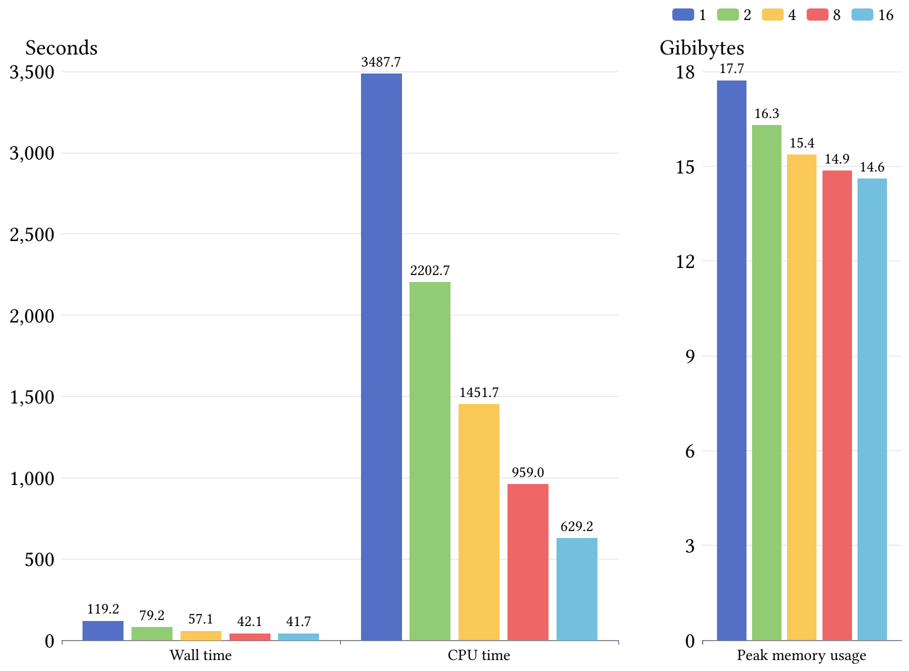

Figure 40: The computational resource metrics for runs of `floxer` with different seed subsampling step sizes. The depicted values for the seed subsampling step size increase exponentially. As the seed subsampling step size increases, both the running time and memory usage of the program decrease. The decrease in the memory usage is small in comparison to the decrease in running time.

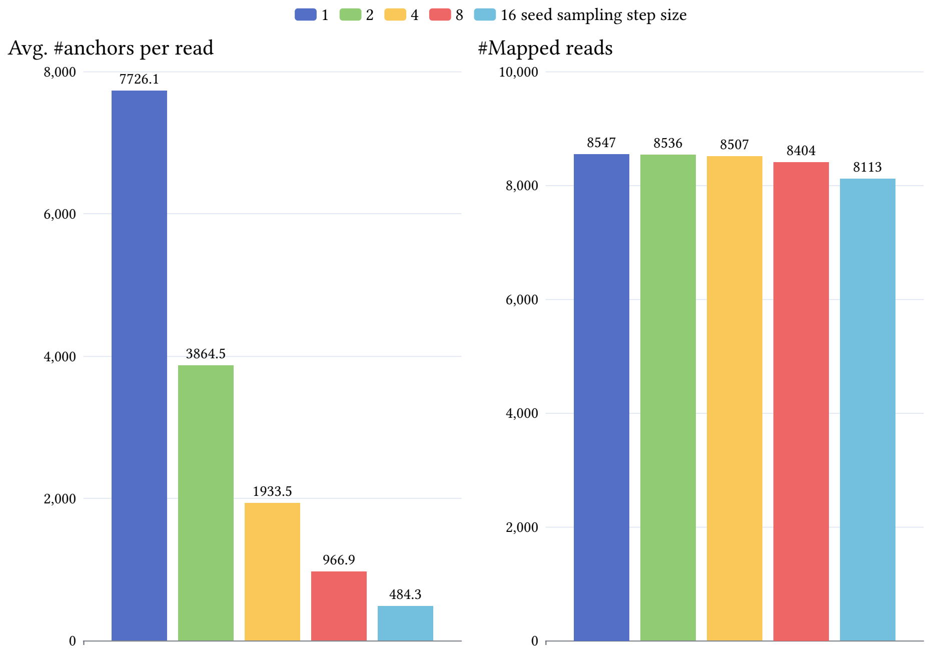

Figure 41: The average number of anchors found per read and number of mapped reads for runs of `floxer` with different seed subsampling step sizes. The depicted values for the seed subsampling step size increase exponentially. The average number of anchors per read decreases in linear terms of the seed subsampling step size. The number of mapped reads on the other hand only decreases slightly.

4.9 Comparison to `minimap2`

This section compares the prototype read mapper of this work, `floxer`, to a state-of-the-art read mapper, `minimap2`. First, the running time and memory usage are compared. Then, a detailed comparison of the alignments returned by the two read mappers is shown. In the benchmarks of this section, `minimap2` has been run with the parameter preset intended for mapping noisy Oxford Nanopore long-reads.

For the performance comparison, `floxer` has been run in three different configurations. In the full output mode with default parameters, in the reduced output mode with default parameters, and in the reduced output mode with *fast* parameters. The fast parameters significantly improve the running time performance of the program, but slightly reduce the number of mapped reads in the output. While `floxer` with the default parameters is able to map 8547 reads, the fast parameters reduce this number to 8361. The fast parameters differ from the default parameters in the following ways:

- The soft anchor cap is set to 10.
- The hard anchor cap is set to 1000.
- The seed subsampling step size is set to 4.

The comparison between `minimap2` and `floxer` in the reduced output mode is not entirely fair, because `minimap2` has to do more work by computing and returning the alignments. However, using the full output mode in the comparison also does not paint an accurate picture of the performance differences between the two methods, because the verification subroutine of `floxer` is not yet fully optimized. Figure 42 shows the computational resources needed to execute `minimap2` and `floxer` in the different configurations on the real data set. In terms of wall time, `minimap2` is about 125 times faster than `floxer` in the full output mode. In terms of user CPU time, the ratio is 23-fold. This difference between the ratios of the wall and CPU time can be explained by the usage of the full output mode for `floxer`, which is based on a not fully optimized verification algorithm. A large part of the bottleneck is the memory allocation for large traceback matrices, which has a detrimental effect on the efficiency of the parallelization of the program. This phenomenon can also be seen in the large amount of system CPU time and memory required by `floxer` in the full output mode compared to `minimap2`.

In the reduced output mode, `floxer` with default parameters takes around 6 times more wall time to run than `minimap2`. Using the fast parameters, `floxer` is only 1.25 times slower. In terms of CPU time, the run of `floxer` with the fast parameters is almost equivalent to `minimap2`. The run with default parameters and the reduced output mode takes about 14 times as many CPU seconds as `minimap2`. The difference in wall and CPU time can be explained by the fact that it has to do more work per read and therefore the critical sections of the multi-threaded implementation are under less contention.

In summary, `floxer` is not competitive in terms of running time and memory usage performance when it has to compute alignments for the full output mode. When taking this step out of the equation by using the reduced output mode, the difference is much closer. When using the fast parameters, `floxer`'s running time and memory usage are comparable to those of `minimap2`.

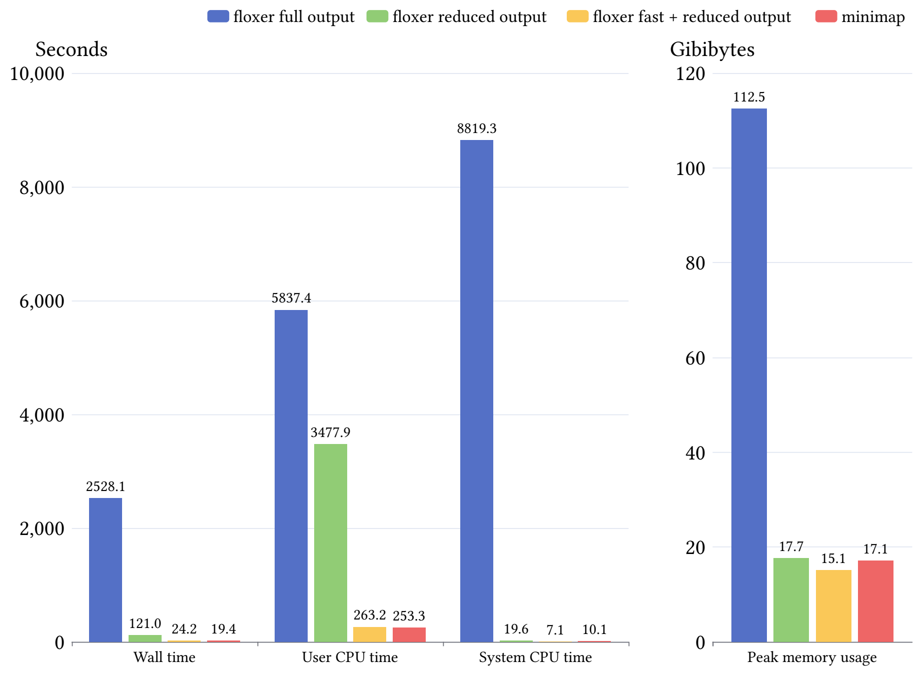

Figure 42: The computational resource metrics for runs of the prototype read mapper of this work, `floxer`, and the state-of-the-art read mapper `minimap2`. While `floxer` in the fast configuration and using the reduced output has a comparable performance to `minimap2`, the other two configurations of `floxer` are not competitive with the state-of-the-art read mapper.

Comparing the results of `floxer` and `minimap2` is not a straightforward task. While `floxer` only searches for read occurrences with a maximum edit distance relative to the read length, `minimap2` has a more advanced model of read occurrences that captures some of the complexity in how reads relate to the reference genome caused by structural variants. To accurately compare the results of the two read mappers, five categories of reads in the output set of the read mappers are distinguished:

- **unmapped** are the reads for which the read mapper was not able to detect any occurrences.
- **linear simple mapped** reads are mapped according to the distance model of `floxer`, i.e. as a continuous sequence and with a maximum edit distance.
- **linear large error rate mapped** are reads that are mapped as a continuous sequence, but their alignments have a higher error rate in terms of edit distance than the maximum configured read error rate of `floxer`.
- **linear large clipping mapped** are reads for which `minimap2` has excluded (clipped) parts on either end of the read. These excluded parts have to be large enough such that if their size multiplied by 0.75 were counted as errors, the total number of errors of the read occurrence would amount to a higher error rate than the configured maximum read error rate of `floxer`. The multiplication by 0.75 is done because even two random DNA sequences are expected to match with a Hamming distance of about 25% of their length.
- **chimeric or inversion mapped** are reads that are partitioned into multiple pieces that match at possibly distant locations in the reference genome. These partitions are modeled by `minimap2` as so-called *supplementary alignments* in the output file. Additionally, this category contains reads mapped by `minimap2` as an inversion structural variant.

If multiple read occurrences belonging to different categories are reported for a single read, the category that is named at the earliest position in the above list is chosen as the category of the read.

Figure 43 shows the categories for different sets of reads. The two leftmost bars show the categories for the whole output sets of `minimap2` and `floxer` on the real data set, respectively. The other bars depict the respective categories of reads that were mapped by both programs, only by `minimap2` and only by `floxer`. For the bar showing the categories of reads mapped by both programs, the category of `minimap2`'s output is depicted.

It can be observed that `floxer` leaves significantly more reads unmapped (44.9%) than `minimap2` (10.3%). None of the reads mapped only by `minimap2` have the linear simple mapped category. The largest category of reads that were mapped by `minimap2` and not `floxer` is the linear large error rate mapped category. It may be possible to detect occurrences of some of these reads by setting the read error rate parameter of `floxer` to a higher value than the default 8%. However, Oxford Nanopore reads typically have an error rate of at most 8%. The large error rate of many reads that were mapped by `minimap2` could be explained by few large insertions or deletions, i.e. structural variants. 12 reads were mapped by `floxer` in the linear large error rate mapped category. These are due to the bottom-up algorithm allowing more errors than configured, up to the number of seed errors (1 in this case) . `floxer` is able to map 69 reads that `minimap2` leaves unmapped. This is about 0.4% of the reads in the whole data set.

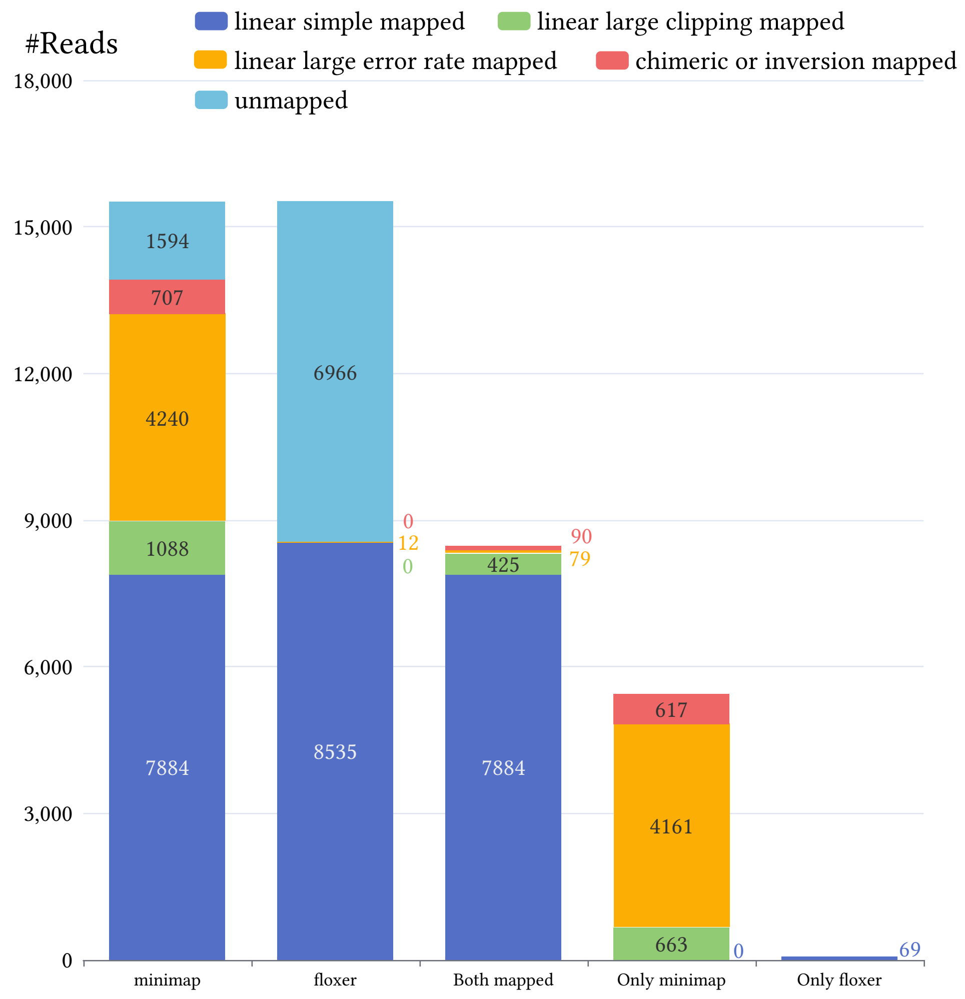

Figure 43: The mapping categories for different sets of reads based on the output of `floxer` and `minimap2`. The two leftmost bars refer to the entire output sets of the two read mappers. The other bars show different logical combinations of the output sets of the two programs. `minimap2` is able to detect occurrences for significantly more reads. Most of the reads mapped only by `minimap2` belong to the linear large error rate mapped category. `floxer` is able to detect an occurrence for a small number of reads that `minimap` leaves unmapped.

5 Discussion

In this thesis, the theoretical groundwork was laid for a novel long-read mapping approach based almost exclusively on non-heuristic methods such as bidirectional FM-Index search and the PEX algorithm. The main idea of the new approach is to not only search for exact occurrences of the pigeonhole seeds of the PEX algorithm, but instead allow a number of errors in possible matches. This can be done with high efficiency by leveraging complete optimum search schemes and the bidirectional FM-Index. Due to the allowed number of errors, the pigeonhole seeds can be increased in length, making them suitable for read mapping applications dealing with large reference genomes.

A first approach based on a minimal change to the PEX tree building algorithm was shown to produce undesirable PEX trees that cannot guarantee the given number of errors for all seeds. To overcome this limitation and allow precise control over the number of seed errors, a completely new bottom-up PEX tree building algorithm was introduced. It was proven that the algorithm generates trees that have the desired properties and guarantee the given number of errors for all seeds.

Furthermore, this work introduced a number of optimizations and algorithmic components that make the application of the PEX algorithm to long-read mapping feasible. Several strategies for reducing the numbers of anchors obtained when searching the seeds in real reference genomes were proposed. These strategies deviate from the goal of exclusively applying exact algorithmic components, but are required for successfully executing the proposed algorithm on a real, biological data set.

An implementation of the novel method was delivered and a wide range of parameters of the program were evaluated. The benchmarks run on a simulated data set showed that the new approach is able to provide significant performance gains over the original PEX algorithm. While the original PEX algorithm was able to detect all read occurrences in reasonable time when applying the soft anchor cap, only the new method was able to run efficiently without the soft anchor cap and therefore guarantee exact results.

When applied to a real, biological data set, the novel approach provided only minimal performance gains over the original PEX algorithm. However, it did lead to a small, but considerable increase in the number of reads for which at least one occurrence could be detected. For large values of the read error rate parameter, the difference in the number of mapped reads was substantially higher, favoring the novel method.

A comparison to the state-of-the-art read mapper `minimap2` showed that the implementation of this work is not yet competitive in terms of running time and memory usage. However, taking the not fully optimized performance bottleneck out of the equation and using ("fast") parameters that slightly reduce the number of mapped reads showed that there is potential to drastically improve the running time and memory usage of the program in the future.

In addition, the prototype read mapper of this work is limited by its inability to detect read occurrences involving structural variants. On a positive note, the new method was able to detect occurrences for a small number of reads that `minimap2` left unmapped.

5.1 Conclusion

By designing, implementing and analyzing the prototype read mapper of this work, a multitude of insights could be gained into the way an exact pattern matching approach like the PEX algorithm can be applied to read mapping problems involving large reference sequences and high read error rates. Searching longer pigeonhole seeds with errors turned out to be an effective solution to the problem of large numbers of random, false positive anchors. However, even within the scope of being a slower and slightly more accurate complementary tool to state-of-the-art read mappers, a considerable amount of future work is required before the prototype can be considered ready for application. Its core limitations are the high usage of computational resources and inability to map long-reads spanning structural variants.

5.2 Future Work

In this section, a number of ideas for future improvements to the algorithm and implementation of the method proposed in this work are listed.

Firstly, it may be possible to further experiment with different combinations of the many parameters of the program. For example, it may be possible to use the `first reported` anchor selection strategy to drastically improve the running time performance of searching with larger numbers of seed errors. In turn, it may be possible to lower the soft anchor cap when using larger numbers of seed errors, which would further reduce the running time. Additionally, an open question that remains from the analysis is how to set the read error rate parameter for a good trade-off between performance and the number of mapped reads.

However, the two main issues of the method in its current state are the non-competitive running time and memory consumption, as well as the inability to handle structural variants. The main issue for the running time is the not yet fully optimized and well-integrated verification subroutine. It would be necessary to apply an algorithm that does not keep the whole traceback matrix in memory at the same time. The running time could be improved by using an implementation based on the ideas of René Rahn on how to leverage modern hardware for fast sequence alignment [50].

To further improve the verification running time, a chaining step could be added that guides the verification procedure. Such a chaining step could partly render the hierarchical verification procedure obsolete. However, a hierarchical verification approach could still be used for short chains or anchors that could not be chained.

Depending on the parameters used, the FM-Index search can be another performance bottleneck of the program. this could be addressed by more closely monitoring and controlling the search procedure of the index. Based on certain statistical parameters of the reference genome and searched seed, more conditions for early aborts of the search could be added. Additionally, a more sophisticated strategy for subsampling seeds could be implemented. One possibility could be an iterative approach where only a small number of seeds are searched in the first step. Then, verifications would immediately be run for the anchors of these seeds. If no adequate read occurrences could be detected, more seeds would be searched and their anchors would be verified. These steps would be repeated until all seeds have been searched or an adequate read occurrence has been found.

Finally, the read mapper prototype of this work needs the ability to detect and handle structural variants in the read occurrences. Otherwise it will not be able to compete with state-of-the-art read mappers in the long-read domain. A first, simple approach of adding support for structural variants to the algorithm could be to collect occurrences of parts of the read that did not result in a full verification in the PEX tree root. Such partial alignments could be reported in the output file by clipping the rest of the read, similar to what `minimap2` does. In addition, it might be possible to explore the idea of joining multiple such partial alignments of a read into a chimeric alignment that covers the whole or most of the read. Another approach would be to apply a chaining algorithm that is aware of the possibility of structural variants before running verifications. This would however be a considerable change to the method and would, to some extent, depart from the goal of exploring a different approach to what most state-of-the-art read mappers use.

Bibliography

[1] L. Mädje, M. Haug, and The Typst Project Developers, “Typst.” [Online]. Available: https://github.com/typst/typst

[2] “DeepL Write.” Accessed: Apr. 22, 2025. [Online]. Available: https://www.deepl.com/en/write

[3] E. S. Lander *et al.*, “Initial sequencing and analysis of the human genome,” *Nature*, vol. 409, no. 6822, pp. 860–921, Feb. 2001, doi: 10.1038/35057062.

[4] J. C. Venter *et al.*, “The sequence of the human genome,” *Science*, vol. 291, no. 5507, pp. 1304–1351, Feb. 2001, doi: 10.1126/1058040.

[5] Wetterstrand KA, “DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP).” Accessed: Jan. 15, 2024. [Online]. Available: https://www.genome.gov/sequencingcostsdata

[6] J. E. Gorzynski *et al.*, “Ultrarapid Nanopore Genome Sequencing in a Critical Care Setting,” *New England Journal of Medicine*, vol. 386, no. 7, pp. 700–702, Jan. 2022, doi: 10.1056/NEJMc2112090.

[7] National Center for Biotechnology Information, “GenBank and WGS Statistics.” Accessed: Jan. 16, 2024. [Online]. Available: https://www.ncbi.nlm.nih.gov/genbank/statistics/

[8] Z. Wang, M. Gerstein, and M. Snyder, “RNA-Seq: a revolutionary tool for transcriptomics,” *Nature Reviews Genetics*, vol. 10, no. 1, pp. 57–63, Jan. 2009, doi: 10.1038/nrg2484.

[9] T. S. Furey, “ChIP–seq and beyond: new and improved methodologies to detect and characterize protein–DNA interactions,” *Nature Reviews Genetics*, vol. 13, no. 12, pp. 840–852, Dec. 2012, doi: 10.1038/nrg3306.

[10] E. Lieberman-Aiden *et al.*, “Comprehensive Mapping of Long-Range Interactions Reveals Folding Principles of the Human Genome,” *Science*, vol. 326, no. 5950, pp. 289–293, 2009, doi: 10.1126/science.1181369.

[11] M. Jain *et al.*, “Nanopore sequencing and assembly of a human genome with ultra-long reads,” *Nature Biotechnology*, vol. 36, no. 4, pp. 338–345, Apr. 2018, doi: 10.1038/nbt.4060.

[12] C. Bleidorn, “Third generation sequencing: technology and its potential impact on evolutionary biodiversity research,” *Systematics and Biodiversity*, vol. 14, no. 1, pp. 1–8, Jan. 2016, doi: 10.1080/14772000.2015.1099575.

[13] K. Sahlin, T. Baudeau, B. Cazaux, and C. Marchet, “A survey of mapping algorithms in the long-reads era,” *Genome Biology*, vol. 24, no. 1, p. 133, Jun. 2023, doi: 10.1186/s13059-023-02972-3.

[14] M. Jain, H. E. Olsen, B. Paten, and M. Akeson, “The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community,” *Genome Biology*, vol. 17, no. 1, p. 239, Nov. 2016, doi: 10.1186/s13059-016-1103-0.

[15] A. McCarthy, “Third Generation DNA Sequencing: Pacific Biosciences' Single Molecule Real Time Technology,” *Chemistry & Biology*, vol. 17, no. 7, pp. 675–676, Jul. 2010, doi: 10.1016/j.chembiol.2010.07.004.

[16] F. Giordano *et al.*, “De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms,” *Scientific Reports*, vol. 7, no. 1, p. 3935, Jun. 2017, doi: 10.1038/s41598-017-03996-z.

[17] N. Stoler and A. Nekrutenko, “Sequencing error profiles of Illumina sequencing instruments,” *NAR Genomics and Bioinformatics*, vol. 3, no. 1, p. lqab19, 2021, doi: 10.1093/nargab/lqab019.

[18] H. Zhang, C. Jain, and S. Aluru, “A comprehensive evaluation of long read error correction methods,” *BMC Genomics*, vol. 21, no. 6, p. 889, Dec. 2020, doi: 10.1186/s12864-020-07227-0.

[19] H. Li, “Minimap2: pairwise alignment for nucleotide sequences,” *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018, doi: 10.1093/bioinformatics/bty191.

[20] H. Li, “New strategies to improve minimap2 alignment accuracy,” *Bioinformatics*, vol. 37, no. 23, pp. 4572–4574, 2021, doi: 10.1093/bioinformatics/btab705.

[21] M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert, “A novel and well-defined benchmarking method for second generation read mapping,” *BMC Bioinformatics*, vol. 12, p. 210, May 2011, doi: 10.1186/1471-2105-12-210.

[22] M. Tarailo-Graovac and N. Chen, “Using RepeatMasker to Identify Repetitive Elements in Genomic Sequences,” *Current Protocols in Bioinformatics*, vol. 25, no. 1, p. 4, 2009, doi: 10.1002/0471250953.bi0410s25.

[23] J. LoTempio, E. Delot, and E. Vilain, “Benchmarking long-read genome sequence alignment tools for human genomics applications,” *PeerJ*, vol. 11, p. e16515, Dec. 2023, doi: 10.7717/peerj.16515.

[24] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with Bowtie 2,” *Nature Methods*, vol. 9, no. 4, pp. 357–359, Apr. 2012, doi: 10.1038/nmeth.1923.

[25] H. Li, “Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM,” Mar. 2013, doi: 10.48550/arXiv.1303.3997.

[26] I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, “Fast and sensitive mapping of nanopore sequencing reads with GraphMap,” *Nature Communications*, vol. 7, no. 1, p. 11307, Apr. 2016, doi: 10.1038/ncomms11307.

[27] F. J. Sedlazeck *et al.*, “Accurate detection of complex structural variations using single-molecule sequencing,” *Nat Methods*, vol. 15, no. 6, pp. 461–468, Apr. 2018, doi: 10.1038/s41592-018-0001-7.

[28] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, “Reducing storage requirements for biological sequence comparison,” *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004, doi: 10.1093/bioinformatics/bth408.

[29] J. Marić, I. Sović, K. Križanović, N. Nagarajan, and M.ˇSikić, "Graphmap2 - splice-aware RNA-seq mapper for long reads," *bioRxiv*, 2019, doi: 10.1101/720458.

[30] C. Jain *et al.*, "Weighted minimizer sampling improves long read mapping," *Bioinformatics*, vol. 36, no. Supplement_1, pp. i111–i118, Jul. 2020, doi: 10.1093/bioinformatics/btaa435.

[31] C. Firtina *et al.*, "BLEND: a fast, memory-efficient and accurate mechanism to find fuzzy seed matches in genome analysis," *NAR Genomics and Bioinformatics*, vol. 5, no. 1, p. lqad4, Mar. 2023, doi: 10.1093/nargab/lqad004.

[32] J. A. C. Ren Mark J. P., "lra: A long read aligner for sequences and contigs," *PLOS Computational Biology*, vol. 17, no. 6, pp. 1–23, 2021, doi: 10.1371/journal.pcbi.1009078.

[33] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences.* Cambridge University Press, 2002, pp. 162–167. doi: 10.1017/CBO9781316135228.

[34] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," vol. 0, no. , pp. 390–398, 2000, doi: 10.1109/SFCS.2000.892127.

[35] M. Burrows, R. W. Taylor, and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," 1994. Accessed: Apr. 17, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:2167441

[36] C. Pockrandt, M. Ehrhardt, and K. Reinert, "EPR-Dictionaries: A Practical and Fast Data Structure for Constant Time Searches in Unidirectional and Bidirectional FM Indices," in *Research in Computational Molecular Biology*, S. C. Sahinalp, Ed., Springer International Publishing, 2017, pp. 190–206. doi: 10.1007/978-3-319-56970-3_12.

[37] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu, "High Throughput Short Read Alignment via Bi-directional BWT," in *2009 IEEE International Conference on Bioinformatics and Biomedicine*, 2009, pp. 31–36. doi: 10.1109/BIBM.2009.42.

[38] G. Kucherov, K. Salikhov, and D. Tsur, "Approximate String Matching using a Bidirectional Index," 2015. doi: 10.1016/j.tcs.2015.10.043.

[39] K. Kianfar, C. Pockrandt, B. Torkamandi, H. Luo, and K. Reinert, "Optimum Search Schemes for Approximate String Matching Using Bidirectional FM-Index," 2018. doi: 10.48550/arXiv.1711.02035.

[40] L. Renders, L. Depuydt, S. Rahmann, and J. Fostier, "Automated Design of Efficient Search Schemes for Lossless Approximate Pattern Matching," in *Research in Computational Molecular Biology*, J. Ma, Ed., Springer Nature Switzerland, 2024, pp. 164–184. doi: 10.1007/978-1-0716-3989-4_11.

[41] S. G. Gottlieb and K. Reinert, "SeArcH schemes for Approximate stRing mAtching," *NAR Genomics and Bioinformatics*, vol. 7, no. 1, p. lqaf25, 2025, doi: 10.1093/nargab/lqaf025.

[42] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational Geometry : Algorithms and Applications*. Springer Berlin / Heidelberg, 2008, pp. 220–226. doi: 10.1007/978-3-540-77974-2.

[43] H. Li *et al.*, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, Jun. 2009, doi: 10.1093/bioinformatics/btp352.

[44] S. G. Gottlieb, "fmindex-collection." Accessed: Mar. 11, 2025. [Online]. Available: https://github.com/SGSSGene/fmindex-collection/releases/tag/v0.5.0

[45] K. Reinert *et al.*, "The SeqAn C++ template library for efficient sequence analysis: A resource for programmers," *Journal of Biotechnology*, vol. 261, pp. 157–168, Nov. 2017, doi: 10.1016/j.jbiotec.2017.07.017.

[46] Heng Li (lh3), "Minimap2-2.28." [Online]. Available: https://github.com/lh3/minimap2/releases/tag/v2.28

[47] Genome Reference Consortium, "Genome assembly GRCh38.p14." Accessed: Mar. 26, 2025. [Online]. Available: https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.40/

[48] N. A. O'Leary *et al.*, "Exploring and retrieving sequence and metadata for species across the tree of life with NCBI Datasets," *Scientific Data*, vol. 11, no. 1, p. 732, Jul. 2024, doi: 10.1038/s41597-024-03571-y.

[49] Nanopore WGS Consortium, "Whole Human Genome Sequencing Project." Accessed: Mar. 26, 2025. [Online]. Available: https://github.com/nanopore-wgs-consortium/NA12878/blob/master/Genome.md

[50] R. Rahn, "Performance-Driven Algorithm Engineering: Optimising Pairwise Sequence Alignment and Pattern Matching Algorithms in the Era of Pangenomic Sequence Analysis," 2023. Accessed: Apr. 24, 2025. [Online]. Available: https://refubium.fu-berlin.de/bitstream/handle/fub188/41976/Dissertation_Rahn.pdf