The Search

Searching for efficient Search Schemes

Simon Gene Gottlieb
simon@gottliebtfreitag.de

Master's Thesis
Freie Universität Berlin
Fachbereich Mathematik und Informatik

Advisor
Sven Bönigk, *Freie Universität Berlin*
Supervisor
Prof. Dr. Knut Reinert, *Freie Universität Berlin*

Berlin, 25.09.2020

Abstract

In bioinformatics, large amounts of genomic data are processed. To analyze this data, algorithms solving the approximate string matching (ASM) problem are used. These allow the identification of genetic variations or sequencing errors, called mismatches. In recent years algorithms based on the concepts of *search schemes* and *optimum search schemes* have been developed. Optimum search schemes are currently only known for 3 or fewer mismatches. This thesis explores the metrics used to find optimum search schemes. To reduce weaknesses a new metric *expected node count* is developed and its improved properties as a performance indicator are demonstrated.

The established metrics identify strategies to construct search schemes that perform similar to *optimum search schemes*. The characteristics from previously known ASM algorithms such as the pigeonhole principle, 01*0 seeds, and suffix filter are used to design heuristics constructing search schemes. Also, known optimum search schemes are investigated to construct a heuristic $H2$ which recreates optimum search schemes and has the lowest node count compared to all other heuristics for any given number of mismatches.

The best performing search scheme depends on the numbers of allowed mismatches, the length of the search pattern and the used distance metric. When using Hamming distance based search schemes, optimized pigeonhole and optimized 01*0 seeds approaches perform best. If the edit distance is used, the search schemes based on suffix filter and $H2$ perform best.

At last, this thesis explores optimizations opportunities. It looks at two different approaches. The first one reduces the number of recursion steps by reducing the possible paths through the algorithm. The second one introduces an additional constraint onto the query by grouping characters together and giving every group an additional distance limitation.

Declaration of Authorship

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Berlin (Germany), 25.09.2020

Simon Gene Gottlieb

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Searching for specific patterns in a text is an old and well-studied field in computer science [9]. The advances in DNA sequencing technology result in a category of search algorithms called *approximate string search*. Matching a large number of snippets of genomic data from sequencing machines to a reference genome is an important part of bioinformatics. This matching must tolerate natural DNA mutations and errors that occur during sequencing while maintaining storage requirements and keeping runtime performance high. Different techniques exist to approach the problem of approximate string search.

Search schemes were first described by Kucherov et al. [10]. Kianfar et al. [8] add a formalization of optimum search schemes. These concepts describe methods to analyze and compare different search schemes. The dissertation by Pockrandt [17] further explores optimum search schemes and compares them with other approximate string search algorithms. While optimum search schemes have shown to have similar or improved runtimes compared to other algorithms, they are only known for search patterns that mismatch in 0-3 characters against a reference text. The number of allowed differences between pattern and text is called the *error count* or *matching errors*. The algorithms used to find optimum search schemes run several days for an error count of 4 and longer for even higher error counts.

1.2 Goal

This thesis explores the possibilities to use existing approximate search strategies and convert them into search schemes. It also examines existing optimum search schemes to find underlying structures to extrapolate these for search schemes with larger numbers of matching errors. Beyond this, it looks at optimizations improving the runtime of the search scheme algorithm. The overall goal of this thesis is to make the construction of well-performing search schemes that allow higher error rates feasible.

1.3 Outline

Chapter 2 takes a look at the background of the current state of research in the field of approximate string search. In the second half, it explains and defines data structures and concepts which form the foundation for all further chapters.

Afterwards Chapter 3 outline a pseudocode implementation. This pseudocode is a simplified implementation of existing search scheme algorithms. Basic optimizations are applied and shown that the resulting algorithm performs similar to existing implementations.

Based on the characteristics seen in the pseudocode implementation Chapter 4 discusses the importance of integrating probabilistic properties as previous work has done. The theoretical performance of search schemes is analyzed by applying a node count metric. This metric is extended by the probabilistic properties and benchmarked, demonstrating that it improves accuracy when used as a performance indicator.

Chapter 5 explores strategies to construct search schemes. Because the computation of optimum search schemes is only feasible for a low number of matching errors, this chapter describes different heuristics to construct search schemes that allow higher numbers of matching errors. It takes different approximate string search algorithms from Chapter 2 and converts them into search schemes. It analyzes known optimum search schemes and derives a constructive method to build search schemes that recreate the known optimum search schemes for low matching errors but also create valid search schemes for higher error rates. This section includes a wide range of performance comparisons between different search schemes operating on different types of queries.

To improve overall runtime performance Chapter 6 takes a look a different optimizations opportunities. The running time of searches based on edit distance is magnitudes higher than searches based on Hamming distance especially when applied to search schemes with larger matching errors. Subsection 6.1 describes optimizations which can be applied for edit distance searches. It exploits the fact that searches based on edit distance find multiple paths through the FM-index. These paths can report the same or close-by position in the reference text multiple times and are considered duplicate results. This chapter explores optimizations to reduce duplicates by reducing the number of paths while guaranteeing that at least one of the followed paths will match. Subsection 6.2 applies additional constraints on the search queries. It suggests a technique that limits certain parts of the query to a lower number of errors. This allows expressing constraints that match the characteristic of certain sequencing machines like lower error count at the beginning of a search query which can increase search performance.

Finally, Section 7 gives an overview of this thesis by summarizing the results of each section. It finishes with possible optimizations techniques and implementation ideas to advance the presented algorithms.

2 Background

The following chapter takes a look at the background needed to understand optimum search schemes. It looks at the research related to approximate string search. It lists the relevant definitions for strings and the FM-index structure, including a bidirectional variant. This chapter closes by looking at algorithms unrelated to FM-index that perform approximate string search. These are used in Chapter 5 to derive heuristics constructing well-performing search schemes.

2.1 Related Work

Newer sequencing methods with higher throughput require improved algorithms that are optimized for the characteristics of analyzing genomic data. Typical approaches use a full-text index. The FM-index suggested by Ferragina and Manzini [3] has proven to be a particularly powerful foundation for further improvements.

Parallel to the development of the FM-index, other concepts, to solve the approximate string search problem, have been explored. Kärkkäinen and Na [7] introduced the concept of a suffix filter that is similar to existing filtration algorithms. These algorithms build on top of a full-text index like the FM-index. They show that suffix filters perform better than previous algorithms. The research of Vroland et al. [21] takes a similar approach. They extend the pigeonhole principle by strategically searching for seeds that match the 01*0 criteria leading to even faster searches.

One early paper focusing on approximate string search and suggesting using it for analyzing genomic data is the paper of Wu and Manber [23]. Many faster algorithms have been proposed over the years. In 1993 Manber and Myers [14] introduced the concept of suffix array. They argue that a suffix array is in practice better for on-line search than a suffix tree which was the leading technique at that time.

Independent of approximate string search, Burrows and Wheeler [1] developed a lossless compression algorithm that uses a novel transformation on the target text. It groups similar suffixes which result in improved compression rates. This transformation is referred by other papers as the *Burrows-Wheeler Transform*.

The idea of suffix array and the Burrows-Wheeler Transform were combined by Ferragina and Manzini [5, 3] into a full-text index that they named FM-index. It uses a suffix array based on the Burrows-Wheeler Transform. The main advantages are reduced space requirements for the index independent of the data while still allowing feasible retrieval time when accessing it as a suffix tree. The FM-index has received many updates and improvements. In 2005 Ferragina and Manzini [4] improved storage requirements

by compressing the data in the underlying structures of the FM-index. The FM-index allows searching only in a forward or backward direction. Lam et al. [11] extend it by synchronizing cursors across two FM-indices enabling bidirectional searches. The concept of optimum search schemes was introduced by Kianfar et al. [8]. They show that optimum search schemes for up to 2 errors can be computed with a mixed integer program (MIP). Pockrandt [17] provides an optimum search scheme for searches with up to 3 errors and a search scheme for 4 errors that uses a reduced number of searches. Running the MIP for more searches takes several days. Computing optimum search schemes for a high number of errors is not feasible with this approach. A practical application using optimum search schemes is described in the article by Wilson et al. [22]. The paper shows the need for search schemes that allow higher error counts to detect off-target locations in connection with CRISPR-Cas9.

2.2 Notation and Definitions

The following chapter specifies the notations used in this thesis.

Definition 1 (String)
Let Σ be a set called an alphabet. Let $T \in \Sigma^n$ be an n-tuple then T is called a string defined over the alphabet Σ. A string T is denoted as $T = (T_0, T_1, ..., T_{n-1})$ where $T_i \in \Sigma$.

The length n of the string T is denoted by $|T|$. A single tuple entry T_i is called *letter* or *character*. This thesis also uses the word *text* for a string. It indicates a string that is magnitudes longer than other strings used in the same context. Also, the terms *query* and *pattern* are being used to refer to strings. The notation $T^{(i)}$ refers to a whole string. If a tuple of strings is given as $(T^{(0)}, T^{(1)}, ...)$ a subscript can be used to refer to the i-th character of all strings.

$$(T^{(0)}, T^{(1)}, ...)_i = (T_i^{(0)}, T_i^{(1)}, ...)$$

For convenience a string is sometimes denoted without parenthesis and without commas between the characters.

$$T = (T_0, T_1, ..., T_{n-1}) = T_0 T_1 ... T_{n-1}$$

Definition 2 (Total order on an alphabet)
Let $<: \Sigma \times \Sigma \to \{0, 1\}$ be a total order on the alphabet Σ.

Definition 3 (Infix)
The notation $T_{i,j}$ refers to an infix of a string forming a $(j - i)$-tuple. This includes all

characters from i-th to the $(j-1)$-th character.

$$T = (T_0, ..., T_i, ..., T_j, ..., T_{n-1})$$
$$T_{i,j} = (T_i, ..., T_{j-1})$$

Note that $T_{0,|T|}$ refers to the complete string. A valid infix $T_{i,j}$ must hold $i \leq j$. If $i = j$ the string is empty.

Definition 4 (Sentinel character)
The sentinel character \$ is a special character added to a given alphabet $\hat{\Sigma}$ forming a new alphabet $\Sigma = \hat{\Sigma} \cup \{\$\}$. The total order is extended to $<: \Sigma \times \Sigma \to \{0, 1\}$ holding $\forall a \in \hat{\Sigma} : \$ < a$.

Definition 5 (Concatenation)
Let R and S be strings. Then $T = R \cdot S$ denotes the concatenation of R and S where $T = R \cdot S = (R_0, R_1, ..., R_{|R|-1}, S_0, S_1, ..., S_{|S|-1})$

Definition 6 (Lexicographical order)
The lexicographical order between two strings $\hat{S}$ and $\hat{T}$ is defined by comparing $S = \hat{S}\$$ and $T = \hat{T}\$$ character-wise until the characters differ. $\hat{S}$ is considered lexicographically smaller than $\hat{T}$ if $\exists i : S_i < T_i \wedge \forall j : j < i \Rightarrow S_j = T_j$.

Definition 7 (Cyclic shift)
A cyclic shift of string T splits the string at character j into $T_{0,j}$ and $T_{j,|T|}$ and concatenates these strings in reversed order. The cyclic shift is denoted by $T^{\ll i} = T_{i+1,|T|} \cdot T_{0,i}$.

A metric is used to measure the distance of two strings. This distance metric is needed to define *approximate* in the context of approximate string search. Wide spread metrics for distances are Hamming [6] and edit [12] distance. Edit distance is also referred to as Levenshtein distance.

Definition 8 (Hamming distance)
Hamming distance measures the distance of two strings A and B of the same length. The distance is the minimum number of characters that have to be substituted in A to result in B. In other words, it measures how many characters differ in A and B. This function will be referred to as $dist_{ham} : \Sigma^n \times \Sigma^n \to \mathbb{N}_0$.

Figure 1 shows an example of the Hamming distance.

Definition 9 (Edit distance)
Applying the edit distance metric does not require two strings of the same length. The

$$\begin{array}{rl} S_1 = ACCGTCG & ACCGTCG \\ S_2 = ACCTTGG & |\ |\ |\ S\ |\ S\ | \\ dist_{ham}(S_1, S_2) = 2 & ACCTTGG \end{array}$$

Figure 1: Hamming distance of two strings S_1 and S_2. Depicting the transformation of S_1 into S_2.

edit distance is the minimum sum of characters that have to be substituted, inserted, and deleted in string A to result in string B. The function $dist_{edit} : \Sigma^n \times \Sigma^m \to \mathbb{N}_0$ returns the edit distance.

Figure 2 depicts an example of the edit distance and two possible alignments.

$$\begin{array}{rlll} S_1 = ACCGTCG & ACCGTCG & ACCGTCG \\ S_2 = ACGTCG & |\ D\ |\ |\ |\ |\ | & |\ |\ D\ |\ |\ |\ | \\ dist_{edit}(S_1, S_2) = 1 & A\ \ CGTCG & AC\ \ GTCG \end{array}$$

Figure 2: Edit distance of two strings S_1 and S_2. Depicting two different possible transformations from S_1 into S_2.

Definition 10 (Edit operation)
An edit operation or short operation transforms a string A into a string B. Typical operations are character *substitution*, *insertion* of a character and *deletion* of a character from a string.

Definition 11 (Edit transcription)
An edit transcription is an ordered set of operations. The size of the ordered set is also called the length of the edit transcript.

A distance metric can also be described by the shortest available edit transcript. For Hamming distance the edit transcript only consists of *substitution* operations allowing to transform a string A into a string B of the same length. For edit distance *substitution*, *insertion* and *deletion* operations are available. Because *insertion* and *deletion* manipulate the length of a given string an edit transcript can transform A into a string B with a different length. In some contexts, it is useful to define a *match* operation which indicates that a character is not being transformed.

Definition 12 (Suffix array)
A suffix array is a data structure that stores the starting positions of all suffixes of a

given text in lexicographical order [14].

Every text T has $|T|$ suffixes since every position in the text is a starting position of a suffix. The function $SA : \mathbb{N}_0 \to \mathbb{N}_0$ maps suffixes based on their ranks to their starting positions in the text. This is a bijective function. Let SA be the suffix array over text T then $SA(i)$ returns the starting position of the i-th ranked suffix. The suffix $SA(i)$ can be denoted with infix notation as $T_{SA(i),|T|}$. Figure 3 shows an example of a suffix array.

idx			SA(idx)	
0	mississippi		10	i
1	ississippi		7	ippi
2	ssissippi		4	issippi
3	sissippi		1	ississippi
4	issippi	sort ⟶	0	mississippi
5	ssippi		9	pi
6	sippi		8	ppi
7	ippi		6	sippi
8	ppi		3	sissippi
9	pi		5	ssippi
10	i		2	ssissippi

Figure 3: Example of a suffix array over the string "mississippi".

The suffix array needs to store every position. For each position $\log(|T|)$ bits are needed. This requires total storage of $O(|T| \cdot \log(|T|))$. It is considered trivial to apply a classical binary search to a suffix array finding a pattern P [14] [13]. The time complexity is $O(|P| \cdot \log(|T|))$. Manber and Myers [14] add auxiliary data structures to improve runtime to $O(|P| + \log(|T|))$. Adding auxiliary data structures improves runtime to $O(|P| + \log(|T|))$ by increasing storage usage.

2.3 FM-Index

The FM-index is a data structure that was developed by Ferragina and Manzini [3, 5]. It is a full-text index that enables a linear search time over P in $O(|P|)$. This section describes the unidirectional FM-index and its extension to a bidirectional version [11].

An FM-index is a data structure build over a text T. Auxiliary data structures constructed from the Burrows-Wheeler Transform enable typical suffix tree operations. The FM-index consists of 4 auxiliary structures: a suffix array $SA : \mathbb{N}_0 \to \mathbb{N}_0$, the Burrows-Wheeler Transform $L : \mathbb{N}_0 \to \Sigma$, an occurrence function $Occ : \mathbb{N}_0 \times \Sigma \to \mathbb{N}_0$ and a count function $C : \Sigma \to \mathbb{N}_0$. Construction and usages are defined in the following

sections.

2.3.1 Burrows-Wheeler Transform

The Burrows-Wheeler Transform of a text is a lossless encoding with beneficial characteristics. In its original variant, it is used as a preprocessing step for data compression [1]. This section shows how the Burrows-Wheeler Transform is applied and how it imitates a suffix array. It forms the foundation for the other FM-index data structures.

Let $\hat{T}$ be a string over an alphabet $\hat{\Sigma}$. Then T is the extended string of $\hat{T}$ with a sentinel character \$ as defined in definition 4. The alphabet Σ is the extended version of $\hat{\Sigma}$ including the sentinel character.

$$T = \hat{T}\$$$
$$\Sigma = \hat{\Sigma} \cup \{\$\}$$

All cyclic shifts $T^{\ll i}$ of the extended text T where $0 \leq i < |T|$ are needed. Since T has a sentinel as a last character $T_{|T|-1} = \$$, it follows that $T^{\ll i}_{|T|-1-i} = \$$. Further, it is known that only one sentinel exists in every $T^{\ll i}$. From this, it follows that two suffixes can not be equal since lexicographical order will always break no later than at the sentinel character $i \neq j \Rightarrow T^{\ll i} < T^{\ll j} \vee T^{\ll i} > T^{\ll j}$. It follows that a unique order of all cyclic shifts $T^{\ll i}$ exists under the lexicographical order. The strings $T^{\ll i}_{0,|T|-i}$ reflect the suffixes of the text T. Let $\mathcal{T}$ be a sorted tuple of the elements in set $\{T^{\ll i} | 0 \leq i < |T|\}$. The tuple $\mathcal{T}$ yields the same order of suffixes as a suffix array over text T as defined in definition 12 with the exception one additional entry at the start of the suffix array covering the added sentinel suffix.

The function $L : \mathbb{N}_0 \to \Sigma$ is the result of the Burrows-Wheeler Transform. It takes the i-th element of $\mathcal{T}$ and retrieves its last character.

$$\mathcal{T} = sort(T^{\ll 0}, T^{\ll 1}, ..., T^{\ll |T|-1})$$
$$L = (\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{|\mathcal{T}|-1})_{|T|-1}$$

Figure 4 shows a visual representation of the construction of the Burrows-Wheeler Transform.

idx	F	L
0	m ississippi	$
1	i ssissippi$	m
2	s sissippi$m	i
3	s issippi$mi	s
4	i ssippi$mis	s
5	s sippi$miss	i
6	s ippi$missi	s
7	i ppi$missis	s
8	p pi$mississ	i
9	p i$mississi	p
10	i $mississip	p
11	$ mississipp	i

$\xrightarrow{sort}$

SA	F	L
11	$ mississipp	i
10	i $mississip	p
7	i ppi$missis	s
4	i ssippi$mis	s
1	i ssissippi$	m
0	m ississippi	$
9	p i$mississi	p
8	p pi$mississ	i
6	s ippi$missi	s
3	s issippi$mi	s
5	s sippi$miss	i
2	s sissippi$m	i

Figure 4: Burrows-Wheeler Transform of $T = mississipi\$$ into $L = ipssm\$pissi$.

2.3.2 Occurrence

The occurrence function $Occ : \mathbb{N}_0 \times \Sigma \to \mathbb{N}_0$ retrieves the number of a specific character in L up to a given position. It counts all occurrences of character c in L between 0 and a position r.

Let $T = \hat{T}\$$ be a string and L is formed over T. Let $c \in \Sigma$ and $0 \leq r < |T|$.

$$Occ(c, r) = \sum_{i=0}^{r} \begin{cases} 1, & L(i) = c \\ 0, & otherwise \end{cases}$$

A trivial implementing of the Occ function has an access time of $O(1)$ and space complexity of $O(|T| \cdot |\Sigma|)$. There exist multiple improved data structures. The seqan3 library uses wavelet trees [18]. These have an access time of $O(\log |\Sigma|)$ but improved space requirements of $O(|T| \cdot \log(|\Sigma|))$. This approach has been improved by Pockrandt et al. [16]. They introduce the concept of EPR-dictionaries. These have the same space requirements of $O(|T| \cdot \log(|\Sigma|))$ but reduces access time to $O(1)$.

2.3.3 Count

In addition to the occurrence function, a count function $C : \Sigma \to \mathbb{N}_0$ is required. It counts the numbers of all characters that have a smaller order than a given character in

T. Let $c \in \Sigma$.

$$C(c) = \sum_{i=0}^{|T|-1} \begin{cases} 1, & L(i) < c \\ 0, & otherwise \end{cases}$$

The space complexity of C is $O(|\Sigma|)$ and access complexity is $O(1)$. Since $|\Sigma|$ is a relatively small number compared to $|T|$, no additional optimizations have to be considered.

2.3.4 LF-Mapping

For the definition of an LF-Mapping, a function $F : \mathbb{N}_0 \to \Sigma$ is required. The function F is similar defined to L but instead of using the last character of the elements of $\mathcal{T}$ it uses the first character. This corresponds to the first column as it can be seen in Figure 4.

$$\begin{aligned} \mathcal{T} &= sort(T^{\ll 0}, T^{\ll 1}, ..., T^{\ll |T|-1}) \\ L &= (\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{|\mathcal{T}|-1})_{|T|-1} \\ F &= (\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_{|\mathcal{T}|-1})_0 \end{aligned}$$

Notice that the result of F is the same as L with all characters in order. This means if F is needed it can be generated from L at any time.

From a previous observation it is known that $T^{\ll i}_{|T|-1} = T_0^{\ll i-1}$. The function $LF : \mathbb{N}_0 \to \mathbb{N}_0$ is defined as a bijective function. Let $rank(T^{\ll i}) = j$ where $\mathcal{T}_j = T^{\ll i}$. Let $r = rank(T^{\ll i})$ and $s = rank(T^{\ll i-1})$. Let $L(r) = T^{\ll i}_{|T|-1}$ and $F(s) = T_0^{\ll i-1}$. The LF-Mapping is defined as $LF(r) = s$.

In practice, this is implemented by using L, Occ, and C.

$$LF(i) = C(L(i)) + Occ(L(i), i)$$

2.3.5 Rank Preserving Property

The rank preserving property of the LF-Mapping is a key feature of the FM-index.

Lemma 1 (rank preserving property)
For all entries in L where $r < s$ and $L(r) \leq L(s)$ it follows that $LF(r) < LF(s)$.

$$\forall r, s : r < s \wedge L(r) \leq L(s) \Rightarrow LF(r) < LF(s)$$

Proof. Let $r = rank(T^{\ll i})$ and $s = rank(T^{\ll j})$

$$
\begin{aligned}
& r < s \wedge L(r) \leq L(s) \\
\Leftrightarrow & rank(T^{\ll i}) < rank(T^{\ll j}) \wedge L(r) \leq L(s) \\
\Leftrightarrow & rank(T^{\ll i}) < rank(T^{\ll j}) \wedge T^{\ll i}_{|T|-1} \leq T^{\ll j}_{|T|-1} \\
\Leftrightarrow & rank(T^{\ll i-1}, \{T\}^{\ll}) < rank(T^{\ll j-1}) \\
\Leftrightarrow & LF(rank(T^{\ll i}) < LF(rank(T^{\ll j}) \\
\Leftrightarrow & LF(r) < LF(s)
\end{aligned}
$$

□

Figure 5 demonstrates the rank preserving property. The arrows do not cross, indicating the correctness of the rank preserving property.

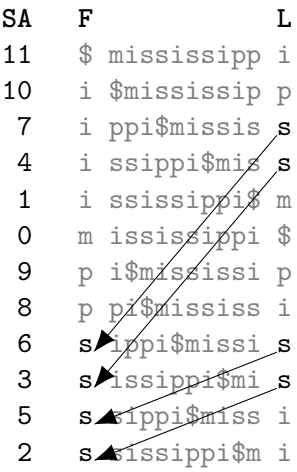

Figure 5: Demonstrating the rank preserving property of the LF-Mapping. The arrows are not crossing.

2.3.6 Locating

Let i be a position in the L function. A lookup in the suffix array SA suffices to find the corresponding position in the text. This has an access time of $O(1)$. Saving the complete suffix array of text T requires $O(|T| \cdot log(|T|))$ bits of memory. Different strategies exist to compress the suffix array SA to reduce memory. This reduction is a trade-off between

space and access time. These compressed suffix arrays are referred to as *sample suffix arrays*. There exists two sampling strategies namely *suffix order sampling* and *text order sampling* which only store $\frac{1}{n}$ of the entries, where n is a parameter. This reduces the memory requirement of both strategies to $O(\frac{|T|}{n} \cdot \log(|T|))$. In case of *suffix order sampling* the average access time is $O(n)$ but the worst-case is $O(|T|)$. The strategy *text order sampling* guarantees to have a worst-case access time of $O(n)$.

Later applications of the FM-index have two positions i and j pointing to a range of suffixes in L. A trivial approach is to iterate over all positions between i and j and calling *locate* for each position. When using a sampled suffix array, more sophisticated locate functions can be used. For example, Cheng et al. [2] suggest applying additional LF-Mapping lookups to i and j instead of applying it to every position. They show that it leads to significantly better runtimes.

2.3.7 Cursor

A cursor $(lb, rb) \in \mathbb{N}_0^2$ is a pair of positions in L where lb and rb form a left and right bound. The left bound is inclusive and the right bound exclusively. Values in L between position lb and rb correspond to a range of suffixes.

$$\forall j : lb \leq rank(T^{\lessdot j}) < rb$$

A cursor marks a range of infix starting positions that match an intermediate pattern. The range of a cursor is empty if $lb \geq rb$. Is the range $(0, |T|)$, it covers all positions of the text.

2.3.8 Backward Step

A backward step takes any character and a cursor and computes a new cursor that covers the previous intermediate pattern extended by the character. Assume a cursor $(lb, rb) \in \mathbb{N}_0^2$ which covers all starting positions of infixes equal to the pattern P. Performing a backward step on (lb, rb) and a character $c \in \Sigma$ will lead to a cursor (lb', rb') that covers all starting positions of infixes that are equal to $P' = c \cdot P$. This function is denoted as $backstep : \mathbb{N}_0^2 \times \Sigma \to \mathbb{N}_0^2$ and is implemented by using the C and Occ function of the FM-index.

$$lb' = C(c) + Occ(c, lb)$$
$$rb' = C(c) + Occ(c, rb)$$

2.3.9 Backward Search

An exact search can be implemented using the *backstep* function. Let (lb_i, rb_i) be the cursor after the i-the iteration. Let P be the search pattern. Let Occ and C be build over text $T = \hat{T}\$$. The algorithm 1 returns a cursor that marks all starting positions of

Algorithm 1 backwardsearch

```
1: function SEARCH(L, Occ, C, P)
2:     (lb_0, rb_0) ← (0, |L|)
3:     for ∀i ∈ ℕ_0 ∧ 0 < i ≤ |P| do
4:         c ← P_{|P|−i}
5:         lb_i ← C(c) + OCC(c, lb_{i−1})
6:         rb_i ← C(c) + OCC(c, rb_{i−1})
7:     return (lb_{|P|}, rb_{|P|})
```

infixes that match the pattern P. To translate the range of the cursor to positions in the text, a locating strategy from Section 2.3.6 has to be applied. The runtime of backward search without locating depends on the length of the pattern. This results in a runtime of $O(|P|)$.

The backward search starts with an empty intermediate pattern hence the starting cursor points to all possible infixes. In each step, the intermediate pattern is extended by prefixing it with a single character from the search pattern P. The cursor is adjusted accordingly. Since the search is done by iterating over the pattern starting with the last character, it is called a backward search.

2.3.10 Forward Search

Similar to the backward search a forward search can be formulated. To achieve forward search the data structures of the FM-index have to be built over the reversed text. Let $T = \hat{T}\$$ be a text and $T_{rev} = (T_{|n|-1}, T_{|n|-2}, T_0)$ be the reversed text of T. Let SA_{fwd}, Occ_{fwd} be defined over T_{rev}.

A forward step is defined equivalently to the backward step. Assume a given cursor (lb_{fwd}, rb_{fwd}) which covers the intermediate pattern P. To compute an updated cursor for pattern $P' = P \cdot c$ a forward step has to be executed.

$$lb'_{fwd} = C_{fwd}(c) + Occ_{fwd}(c, lb_{fwd})$$
$$rb'_{fwd} = C_{fwd}(c) + Occ_{fwd}(c, rb_{fwd})$$

Notice that the cursor is pointing at the ends of the infixes.

2.3.11 Bidirectional Search

Backward and forward searches have separate cursors. It is not possible to take cursors of unidirectional FM-indices covering an intermediate pattern and extend these arbitrarily to the left or right. This limitation is overcome by the bidirectional search introduced by Lam et al. [11]. It combines the forward and backward indices allowing to extend an intermediate pattern in both directions. Let $(Occ_{bwd}, C_{bwd}, SA_{bwd})$ be the FM-index over a text T. Let $(Occ_{fwd}, C_{fwd}, SA_{fwd})$ be the forward FM-index over text T_{rev} which is used to apply forward steps. Let (lb_{fwd}, rb_{fwd}) be a cursor for the forward index and (lb_{bwd}, rb_{bwd}) a cursor for the backward index.

Notice that $C_{bwd} = C_{fwd}$ since the total number of each character is the same in the text and in the reversed text. Let $C = C_{bwd} = C_{fwd}$.

Since a backward step is adjusting the cursor, such that the intermediate pattern is extended to the left, the function is called $extend\_left$. It consists of two parts where the first part is equal to the *backstep*.

$$\begin{aligned} lb'_{bwd} &= C(c) + Occ_{bwd}(c, lb_{bwd}) \\ rb'_{bwd} &= C(c) + Occ_{bwd}(c, rb_{bwd}) \end{aligned}$$

The second part synchronizes the forward cursor so that (lb'_{fwd}, rb'_{fwd}) is a valid cursor that marks the same infixes as (lb'_{bwd}, rb'_{bwd}).

$$\begin{aligned} lb'_{fwd} &= lb_{fwd} + \sum_{j<c} C(j) + Occ_{bwd}(j, rb_{bwd}) - (C(j) + Occ_{bwd}(j, lb_{bwd})) \\ &= lb_{fwd} + \sum_{j<c} Occ_{bwd}(j, rb_{bwd}) - Occ_{bwd}(j, lb_{bwd}) \\ rb'_{fwd} &= lb'_{fwd} + (rb'_{bwd} - lb'_{bwd}) \end{aligned}$$

The function $extend\_left$ is shown in the pseudocode 2. A forward step $extend\_right$ can be implemented equivalently to $extend\_left$.

Definition 13 (Bidirectional cursor)
The cursors (lb_{bwd}, rb_{bwd}) and (lb_{fwd}, rb_{fwd}) can be combined into one bidirectional cursor. A trivial approach would be to combine both cursors into a 4-tuple.

$$cursor_{bi-triv} = (lb_{fwd}, rb_{fwd}, lb_{bwd}, rb_{bwd})$$

This can be improved by combining them into a joined bidirectional cursor $cursor_{bi} \in \mathbb{N}_0^3$. From the previous equations it follows that a bidirectional cursor always holds $rb_{fwd} - lb_{fwd} = rb_{bwd} - lb_{bwd}$ indicating that each cursor has the same range. A simpler version of the bidirectional cursor is defined as a triplet of the left bounds and the range.

$$cursor_{bi} = (lb_{fwd}, lb_{bwd}, range)$$

If the right bounds are needed they can be computed by $rb_{fwd} = lb_{fwd} + range$ and $rb_{bwd} = lb_{bwd} + range$.

Another optimization can be done by dismissing one of the suffix arrays SA_{bwd} or SA_{fwd}. Only one of them is needed for the locating step. A bidirectional FM-index can be defined as $(Occ_{bwd}, Occ_{fwd}, C, SA_{bwd})$.

Algorithm 2 Extending bidirectional cursor to the left

```
1: function EXTEND_LEFT(c, cursor)
2:     (lb_bwd, lb_fwd, range) ← cursor
3:     lb'_bwd ← C(c) + OCC_BWD(c, lb_bwd)
4:     range' ← OCC_BWD(c, rb_bwd) − OCC_BWD(c, lb_bwd)
5:     prev ← 0
6:     for ∀σ ∈ Σ : σ < c do
7:         prev ← prev + OCC_BWD(σ, rb_bwd) − OCC_BWD(σ, lb_bwd)
8:     lb'_fwd ← lb_fwd + prev;
9:     return (lb'_bwd, lb'_fwd, range')
```

2.4 Approximate String Search

Matching strings is a common problem. Let P be a pattern and T a text. The string-matching problem is specified as finding all infixes of T that match P [23]. It can be expressed as finding the set $R = \{i | P = T_{i,i+|P|}\}$.

This can be extended to the approximate string search problem that is not limited to exact matches but includes an allowed distance k between the pattern P and the infixes of T. For Hamming distance, the problem can be expressed as finding the set R_{ham}.

$$R_{ham} = \{i | dist_{ham}(P, T_{i,i+|P|}) \leq k\}$$

The edit distance needs to consider the possibility of different string lengths leading

back to the problem of finding the set R_{edit}.

$$R_{edit} = \{i | \exists j : -k \leq j \leq k \rightarrow dist_{edit}(P, T_{i,i+|P|+j}) \leq k\}$$

The following subsections take a look at different algorithms that solve the approximate string search for Hamming and edit distance.

2.4.1 Backtracking

In 1993 Ukkonen [19] proposed to traverse an indexed text to implement an approximate search based on edit distance allowing for k or fewer errors. In this approach, the indexed text is represented by a suffix tree. Previous approaches [15] already focused on suffix tree construction, but only implemented exact search methods. Each path to any node that starts in the suffix tree root is a possible search. The difficulty is to correctly traverse the tree according to a given search query.

Ukkonen extended these approaches to approximate string searches. Instead of naively searching for every string that is in a given edit distance of the query it groups common suffixes together. This approach is called backtracking and corresponds to a depth-first search through the suffix tree.

Instead of using a suffix tree, it is possible to use an FM-index [17]. This has the advantages of relying on the newest FM-index construction methods and storage techniques.

Figure 6 shows an example of an approximate search for $P = ACGT$ over the alphabet $\Sigma = \{ACGT\}$. The shown diagram applies a Hamming distance for up to 1 error. The edges represent one step of the backtracking algorithm. Solid edges indicate matches with the query whereas dashed edges indicate substitutions. Because the example only allows up to 1 error, all paths to the leaves have a maximum of one dashed line.

A simplified and generalized version of this is shown in Figure 7. It also shows an approximate search with a Hamming distance for up to 1 error. Instead of a concrete search query, it is applicable for all queries with length 4 with any type of alphabet. The substitution branches are collapsed into one single substitution branch. This simplifies the visual representation.

2.4.2 Pigeonhole Principle

The pigeonhole principle states that distributing n objects to m boxes will lead to one box with at least $\lceil \frac{m}{n} \rceil$ objects. This can be also reformulated to distributing n objects

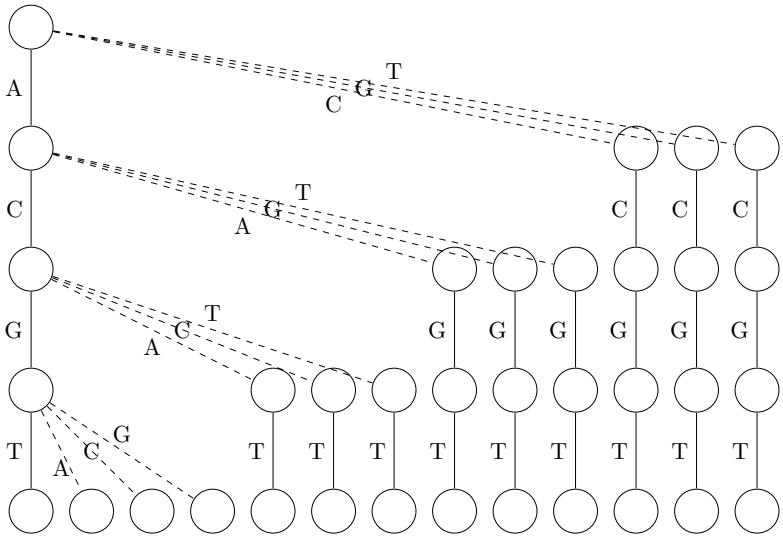

Figure 6: Backtracking of search "ACGT" with maximum 1 error applying Hamming distance. Represents visited paths of a suffix tree. Solid lines indicate exact matches whereas dashed lines show substitutions.

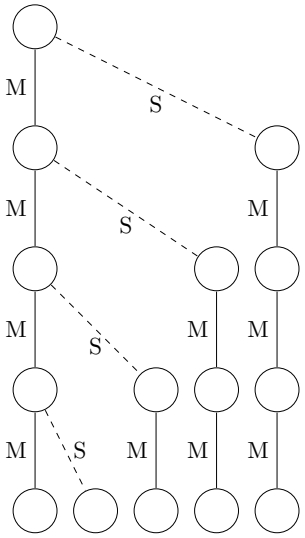

Figure 7: Shows a general scheme for backtracking for queries of length 4 and with maximum 1 error using Hamming distance. Solid lines marked with an "M" represent exact matches. Dashed lines marked with "S" stand for all substitution possibilities.

to m boxes which leads to one box with most $\lfloor \frac{m}{n} \rfloor$ objects. An even stricter specification concludes that if $n < m$ it follows that one of the boxes must be empty.

The principle can be applied to the approximate string search problem and be used

in combination with a bidirectional FM-index [17]. Pockrandt shows that this approach outperforms backtracking in most cases. Assume an approximate search with up to k errors. Divide the query P into $k+1$ chunks. From the pigeonhole principle, it follows that at least one chunk has 0 errors. The backtracking approach iterating over the search pattern once and traversing the FM-index finds all matches. In contrast, the pigeonhole principle iterates multiple times over the search pattern. Each time a different chunk is used as a starting point. The first chunk of every search is searched as an exact match allowing 0 errors. The following chunks are searched as in the backtracking approach with k errors. Every search finds only a partial set of matches.

Applying this algorithm to a query of length 4 and allowing 1 error leads to a general scheme as shown in Figure 8. This algorithm will lead to the same result set as the backtracking algorithm depicted in Figure 7. The key difference is that instead of one search tree the pigeonhole principle has one search tree per chunk. Branching of mismatches only appears in the lower part of the tree which leads to a smaller number of nodes. Note that in the generalized scheme each branch stands for all substitution possibilities which increases with larger alphabets.

One disadvantage of this algorithm compared to backtracking is that the results are being reported multiple times. If duplicates free results are required, a post-processing step has to be implemented.

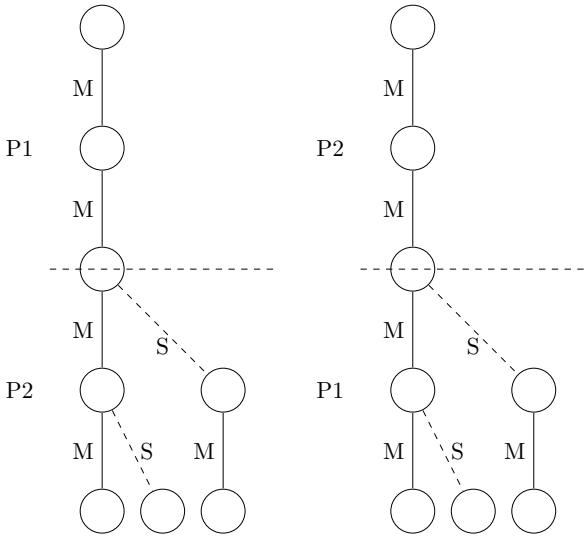

Figure 8: Shows searches for the pigeonhole principle. A query of length 4 and with maximum 1 error using Hamming distance is used. Solid lines marked with an "M" represent exact matches. Dashed lines marked with "S" stand for all substitution possibilities.

2.4.3 01*0 Seeds

Vroland et al. suggest to search for 01*0 seeds and apply in-text verification [21]. Pockrandt [17] points out that it is possible to use bidirectional FM-index to replace the in-text verification step and complete the full search using the FM-index. The 01*0 principle can be seen as an extension of the pigeonhole principle. The main idea is to search for patterns with two chunks with no errors which are separated by an arbitrary number of chunks that have exactly one error. Vroland et al. show in their paper that these are lossless seeds, which means these seeds will find all matches with a maximum distance of k.

Similar to the pigeonhole principle, if n objects are distributed to $n+2$ ordered boxes at least two boxes are empty. Since the boxes are ordered, the 01*0 principle also states that one pair of empty boxes exists which are only separated by boxes with only 1 object inside. This is a much stronger limitation than the pigeonhole principle.

Applying this for approximate string search with k errors the query is divided into $k+2$ chunks. This guarantees that there exist at least two chunks with 0 errors that are separated by chunks that have exactly 1 error.

Figure 9 visualize the same case for 01*0 as it was done before for pigeonhole and backtracking. It shows a generalized search scheme for a query of length 4 with applying Hamming distance. Notice that the search query chunks are not equally sized. This is because a search of length 4 can not be divided equally into 3 chunks. In this particular case, the front chunks are one larger than the back chunks. This is an arbitrarily chosen partition any other partition can be used. It shares the same disadvantage as the pigeonhole principle in finding matches multiple times requiring a post-processing step to remove duplicates.

2.4.4 Suffix Filter

Suffix filters are introduced by Kärkkäinen and Na [7]. They belong to the class of factor filters. Similar to the 01*0 principle this approach searches for seeds that are validated by in-text verification.

When performing an approximate search with k errors the query is split into $k+1$ chunks. From the pigeonhole principle, it is known that at least one chunk has 0 errors. The algorithm searches for a chunk with 0 errors and unidirectional extends the search pattern chunk-wise as long as the strong match criterion is satisfied. Kärkkäinen and Na refer to these chunk as factors. Informally the strong match criterion requires that there are never more errors than chunks. To formalize this criterion a definition for

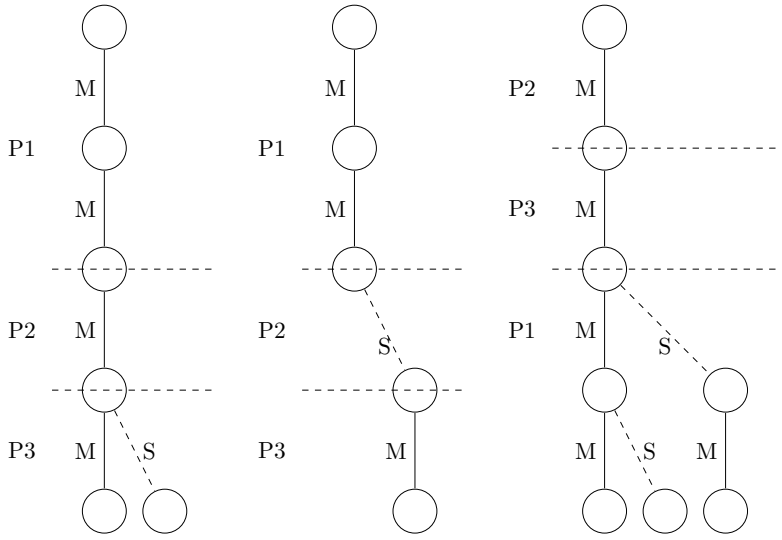

Figure 9: Shows searches for 01*0 principle. A query of length 4 and with maximum 1 error using Hamming distance is used. Searches 1 and 2 have the same order of chunks and could be executed in one search. Solid lines marked with an "M" represent exact matches. Dashed lines marked with "S" stand for all substitution possibilities.

factorization of strings is needed.

Definition 14 (String factors)
Let assume string A is divided into chunks, then there must exist a partition of B such that

$$dist(A, B) = \sum dist(A_i, B_i)$$

The chunks A_i and B_i can be weighted by a corresponding t_i. This allows making the following assumption about factor filters.

Definition 15 (Factor filters)
If $dist(A, B) < \sum t_i$ then there exists a value t_i such that $dist(A_i, B_i) < t_i$.

Kärkkäinen and Na suggest picking $\sum t_i = k + 1$ to get the strongest filter. In this thesis t_i will always be 1 and the number of chunks will be $k + 1$. Since the partition of A can be chosen in any way, the challenge is to find a partition for B. For this, a definition of *match on interval* and *strong match on interval* is needed which supports the search for a valid partition of B.

Definition 16 (Match on interval)
If neighboring chunks of A and B have a smaller distance than the number of chunks

the strings match on an interval.

$$dist(A_{i,j}, B_{i,j}) < j - i$$

Definition 17 (Strong match on interval)
It is also possible to make an even stronger assumption where not only neighboring chunks match on an interval but all prefixes also have to match on the interval. If this is the case, this is considered a strong match.

$$\forall k : i < k \leq j \rightarrow dist(A_{i,k}, B_{i,k}) < k - i$$

Kärkkäinen and Na show that a partition of string B with at least one strong match exists. They applied this approach to find seeds and then perform in-text verification. Similar to Pockrandt's method of extending 01*0 seeds to a bidirectional FM-index it is possible to do the same with suffix filters. For this, a query gets split into $k+1$ chunks. Every chunk is the starting point for a search. A chunk is searched with 0 errors and then extended to the right allowing 1 additional error for every added chunk. If all chunks are added to the right, the rest of the pattern is extended allowing all k errors.

Figure 10 shows a generalized search scheme for a query of length 4 applying Hamming distance. It shares the same disadvantages as the previous methods that find matches multiple times requiring a post-processing step to remove duplicates.

2.5 Search Schemes

The previously introduced search principles describe search steps in a very specific non-general form. This makes theoretical comparisons between them very difficult.

In 2016 Kucherov et al. [10] introduced the concept of search schemes. They provide a formal description and analysis of search schemes and their performance. A search scheme consists of multiple searches where each search is traversing parts of the bidirectional FM-index. The overall idea is to partition the search query in multiple chunks or pieces. Each search describes in which order to search for these pieces. It gives every piece an individual accumulated error bound which limits the number of errors that are accepted in a piece but also a minimum number of errors that have to be found. Splitting queries into multiple pieces and limiting the error count for each piece is similar to the pigeonhole principle where the first chunk has a specific limitation. Search schemes extend this to an individual limit for each piece.

Definition 18 (Search schemes)

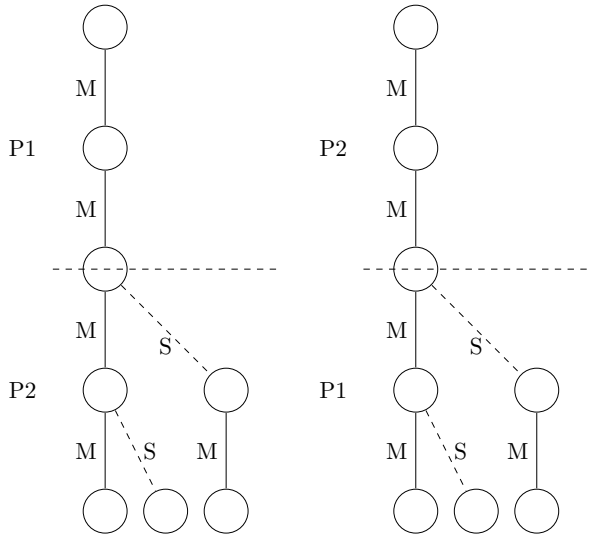

Figure 10: Shows searches applying suffix filters. A query of length 4 and with maximum 1 error using Hamming distance is used. Solid lines marked with an "M" represent exact matches. Dashed lines marked with "S" stand for all substitution possibilities.

Let $S = \{s^{(0)}, s^{(1)}, ..., s^{(n-1)}\}$ be a search scheme. A search query is divided into $n \in \mathbb{N}_0$ number of pieces. Let $s^{(i)} = (\pi^{(i)}, L^{(i)}, U^{(i)})$ denote a single search where $\pi^{(i)} \in \mathbb{N}_0^n$ denotes the order of pieces, $L^{(i)} \in \mathbb{N}_0^n$ denotes the lower bound and $U^{(i)} \in \mathbb{N}_0^n$ denotes the upper bound of accumulated errors of each piece.

A search scheme must fulfill certain properties to be valid. The entries of $\pi^{(i)}$ must meet the connectivity property. A bidirectional search allows extending an intermediate search pattern to the left or the right. Meaning that $\pi_r^{(i)}$ must be a neighbor of any previous piece.

Definition 19 (Connectivity property)

$$\forall i : 0 < r < n \Rightarrow \pi_r^{(i)} \in \{\max_{s<r} \pi_s^{(i)} + 1, \min_{s<r} \pi_s^{(i)} - 1\}$$

Example Expressing the backtracking algorithm as a search scheme is simple. It consists of one search which keeps the query in one piece with a lower bound of 0 and an upper bound of k errors.

$$\mathcal{S}_{bt,k} = \{((0), (0), (k))\}$$

2.5.1 Expanding Search Schemes

A key component to analyze the performance of search schemes is the expansion step which adjusts the lower and upper bound of a search scheme to the size of a specific query length.

The work of Kucherov [10], Kianfar [8] or Pockrandt [17] uses an expansion as part of their performance analysis.

I will show that the analysis of the expansions in the mentioned work is not correct or incomplete and specify a corrected formula that works for searches that apply edit distance and how it can be optimized for Hamming distance. These corrections are not only important for performance analysis. In the implementation in Chapter 3, I will use this expansion to implement a search scheme based search. This differs from previous implementations that operate on the search scheme bounds L instead of the expanded bounds.

Some more notation for search schemes is needed. A given lower bound $L^{(i)} = (0, 0, 1)$ encodes that a search query is divided into 3 pieces. In the following the lower bound will be denoted in a short form $L = 001 \Leftrightarrow L^{(i)} = (0, 0, 1)$. The same notation rules apply for $U^{(i)}$ and $\pi^{(i)}$. Let $n = |P|$ be the number of characters then $E \in \mathbb{N}_0^n$ represents the number of operations that might be applied to, behind, or in front of each character to find a match. This is considered an error distribution. To analyze performance L must be expanded into an $\mathcal{L}$ where the length of $\mathcal{L}$ must match the length of a given pattern P. The upper bound U has to be expanded to $\mathcal{U}$.

Proof. expansion Kucherov et al. fails. In Kucherov et al. work they give an example that partitions the search query into three pieces with characters evenly distributed. They transform $L = 001$ into $\mathcal{L} = 000011$. While L would meet the requirement to cover $E = 000001$ the expanded lower bound $\mathcal{L}$ does not cover $E = 000001$. □

Proof. expansion Pockrandt fails. Similar to Kucherov et al. Pockrandt's work gives an example $L = 0012$ which they expand to $\mathcal{L} = 00001122$. A query with an error pattern $E = 00000101$ would be covered by L, but $E = 00000101$ is not covered by $\mathcal{L}$. □

Kianfar introduces a more sophisticated rule for expansion. But this rule is limited to Hamming distance searches. The expanded lower bound will fail for searches using edit distance. Also, it misses an optimization opportunity on the lower boundary for Hamming distance. The previous works used $\mathcal{L}$ and $\mathcal{U}$ only as a performance indicator. The shown mistakes only lead to minor counting errors. This thesis presents a minimal version of the search algorithm as pseudocode using $\mathcal{L}$ and $\mathcal{U}$ instead of L and U. Having

a correct expansion step is mandatory for a properly functioning algorithm. To simplify the expansion formula a helper function $block(i)$ is defined. It maps the position of an $\mathcal{L}_i$ or $\mathcal{U}_i$ to the corresponding index of L_j and U_j.

Definition 20 (Expansion map)

$$block(i) = \lceil l \cdot |P|/i \rceil$$

With the expansion map the expansion of L, U to $\mathcal{L}^{(edit)}, \mathcal{U}^{(edit)}$ can be defined.

Definition 21 (expanded $\mathcal{L}^{(edit)}$, $\mathcal{U}^{(edit)}$)

$$\mathcal{L}_i^{(edit)} = \begin{cases} 0, & i \leq |q| \wedge block(i) = 0 \wedge block(i+1) = 0 \\ L_{block(i)-1}, & i \leq |q| \wedge block(i) \neq 0 \wedge block(i) = block(i+1) \\ L_{block(i)}, & otherwise \end{cases}$$

The upper limit $\mathcal{U}$ can be determined the same way as in Kucherov et al. work [10] by repeatedly copying the values from U.

$$\mathcal{U}_i^{(edit)} = U_{block(i)}$$

The bounds $\mathcal{L}^{(edit)}$ and $\mathcal{U}^{(edit)}$ are also valid for Hamming distance searches. It is possible to transform $\mathcal{L}^{(edit)}$ and $\mathcal{U}^{(edit)}$ into corresponding $\mathcal{L}^{(ham)}$ and $\mathcal{U}^{(ham)}$ which have stricter bounds. The key observation is that in Hamming distance only one error for each character can occur since only substitutions errors are possible. This means that the difference between $\mathcal{L}_i^{(ham)}$ and $\mathcal{L}_{i+1}^{(ham)}$ can not be greater than 1. Similar constraints exists for $\mathcal{U}^{(ham)}$.

Definition 22 (expanded $\mathcal{L}^{(ham)}, \mathcal{U}^{(ham)}$)

$$\mathcal{L}_i^{(ham)} = \max(\mathcal{L}_{i+1}^{(ham)} - 1, \mathcal{L}_i^{(edit)})$$
$$\mathcal{U}^{(ham)}(i) = \min(\mathcal{U}_{i-1}^{(ham)} + 1, \mathcal{U}_i^{(edit)})$$

2.5.2 Performance Analysis

Given an error count k there are many options to construct a search scheme. To estimate the performance an indicator called the *node count* is used. When representing a search by its paths through the suffix tree, every intermediate result is represented as a node. Assuming that every represented node is traversed, the node count is reflecting the runtime. For a search scheme to be efficient a small number of nodes is required [8, 10].

To count nodes a concrete length of a search pattern has to be assumed. The parameter L and U of each search are being expanded to $\mathcal{L}$ and $\mathcal{U}$. Depending on the applied distance metric the computation of the node count is different.

Definition 23 (Node count - Hamming distance)

$$nc_{ham}(|P|) = \sum_{d=\mathcal{L}_{|P|-1}}^{\mathcal{U}_{|P|-1}} n_{|P|-1,d}$$

$$n_{l,d} = \begin{cases} n_{l-1,d} + (\sigma - 1) \cdot n_{l-1,d-1} & l \geq 1 \wedge \mathcal{L}_l \leq d \leq \mathcal{U}_l \\ 1 & l = 0 \wedge d = 0 \\ 0 & otherwise \end{cases}$$

Definition 24 (Node count - edit distance)

$$nc_{edit}(|P|) = \sum_{d=\mathcal{L}_{|P|-1}}^{\mathcal{U}_{|P|-1}} n_{|P|-1,d}$$

$$n_{l,d} = \begin{cases} n_{l-1,d} + (\sigma - 1) \cdot n_{l-1,d-1} + \sigma \cdot n_{l,d-1} + n_{l,d-1} & l \geq 1 \wedge d \geq 1 \wedge \mathcal{L}_l \leq d \leq \mathcal{U}_l \\ 1 & l = 0 \wedge d = 0 \\ 0 & otherwise \end{cases}$$

Besides the *node count*, search schemes have some more properties. One important attribute is their *completeness* which indicates that they cover all error distribution. A second property is that the searches are *disjoint* meaning an error distribution is only found by one search of the search scheme. It is required that search schemes are complete. This means when searching for a pattern every position will be found by at least one search. It also seems preferable to have search schemes that are *disjoint* to achieve smaller node counts.

2.5.3 Optimum Search Schemes

The *node count* of search schemes is used to define optimum search schemes [8].

Definition 25 (Optimum search scheme)
An optimum search scheme is a search scheme that covers all patterns (*completeness*) with a given number of allowed errors with a minimum node count.

This means there exists no search scheme with a lower node count. A different search scheme may exist with the same node count. This search scheme would also be considered an optimum search scheme.

Pockrandt [17] states that searching for optimum search schemes with a *mixed integer program* (MIP) for error counts larger than 3 takes several days. It seems that this approach is not feasible for higher error numbers.

3 Pseudocode Implementation

Many works [17, 8, 10] have a description of the search algorithm using search schemes. Most of them give pseudocode for the backtracking algorithm and a textual description for the search scheme algorithm. In this section, I want to give a complete pseudocode implementation for the search scheme algorithm.

Existing implementation using search schemes in libraries as *seqan2* and *seqan3* use π, L and U as an input. The presented pseudocode will work on the expanded version Π, $\mathcal{L}$ and $\mathcal{U}$ where Π will be introduced in the next chapter. This enables a more compact version of the algorithm, ease discussion, and is hopefully the basis for more optimizations and greater performance beyond this thesis.

3.1 Expanding π

The expanded version of π is denoted as $\Pi \in \mathbb{N}_0^{|P|}$ and works similar to π. Every entry of Π covers exactly one character of a search pattern P. It reorders characters instead of chunks. When expanding, the connectivity property has to be preserved.

Definition 26 (Expanded Π)

$$block\_size(i) = \begin{cases} \lfloor \frac{|P|}{|L|)} \rfloor + 1, & block(i) < (|P| \mod |L|) \\ \lfloor \frac{|P|}{|L|)} \rfloor, & otherwise \end{cases}$$

$$start\_pos = \sum_{j=0}^{j<|L|} \begin{cases} block\_size(j), & \pi_j < \pi_0 \\ 0, & otherwise \end{cases}$$

$$\Pi_i = \begin{cases} start\_pos, & i = 0 \\ max_{j<i}(\Pi_j) + 1, & i \neq 0 \wedge (block(i) = 0 \vee \pi_{block(i)-1} < \pi_{block(i)}) \\ min_{j<i}(\Pi_j) - 1, & i \neq 0 \wedge (\pi_{block(i)-1} > \pi_{block(i)}) \end{cases}$$

The function *block_size* computes how many characters each piece of the search scheme is covering. It also determines the partition of the search pattern into pieces and has to match the partitioning for L and U. The *block_size* is used to specify *start_pos* which indicates how many characters are covered by pieces in front of π_0. The expansion of π_0 always starts with *start_pos* with increasing numbers. Every following π_i expansion depends on the previous π_{i-1}. This ensures that the connectivity property is upheld.

As an example, let $\pi = 01234$ and $|P| = 9$, this leads to an expansion of $\Pi =$

012345678. Another example, let $\pi = 34210$ and $|P| = 9$, this leads to an expansion of $\Pi = 678543210$.

3.2 Search Algorithm

It is assumed the bidirectional FM-index over a given text $T = \hat{T}\$$ is constructed in a preprocessing step. Let $\Sigma = \hat{\Sigma} \cup \{\$\}$ be the alphabet of T. Let $(Occ_{bwd}, Occ_{fwd}, C, SA_{bwd})$ be the constructed FM-index over T. Let P be the search string over the alphabet $\hat{\Sigma}$. Let $(\Pi, \mathcal{L}, \mathcal{U})$ be a single search from a search scheme.

The algorithm 3 declares which variables are globally accessible and initializes variables used in the recursion step. Every call to algorithm 4 checks if a cursor has reached the termination criteria or processes one of 4 possible edit operations. The direction in which a pattern has to be extended is done in algorithm 5 while the actual extension is done as described in algorithm 2.

Algorithm 3 Main entry point

```
1: function MAIN((Π, L, U), P, T, index)
2:     global Π, L, U, P, T, index
3:     e ← 0
4:     pos ← 0
5:     cursor ← (0, 0, |T|)

6:     SEARCH(cursor, e, pos)
```

3.3 Optimization

The algorithm 4 from Chapter 3.2 comes without optimizations. In this section, I briefly go over two optimizations that are being used when benchmarking and comparing this pseudocode to the implementation in the seqan3 library. In Section 6, I will look at more sophisticated optimizations.

Optimizing Hamming distance The first optimization is more an adjustment to the algorithm such that it is applicable for Hamming distance based searches. While the edit distance metric allows substitutions, insertions, and deletions of characters, Hamming distance metric only allows substitution. To achieve this the lines 16-20 from the algorithm 4 can be removed. This also allows to tighten the abort-condition in line 7. All adjustments can be seen in algorithm 6.

Algorithm 4 Single search step

```
1:  function SEARCH(cursor, e, pos)
2:      if range(cursor) = 0 then return                ▷ abort on empty cursor
3:      if pos = len(q) then                                    ▷ report condition
4:          if L[pos − 1] ≤ e ≤ U[pos − 1] then
5:              REPORT(cursor)
6:          return
7:      if U[pos] < e then return                       ▷ abort on out of bound
8:      α ← q[Π[pos]]                          ▷ retrieve expected next character
9:      for ∀σ ∈ Σ̂ do                  ▷ compute cursor extended by one character
10:         new_cursor[σ] ← EXTEND(σ, cursor)
11:     if L[pos] ≤ e then                                      ▷ search match
12:         SEARCH(new_cursor[α], e, pos + 1)
13:     if L[pos] ≤ e + 1 ≤ U[pos] then                 ▷ search substitution
14:         for ∀σ ∈ Σ̂ ∧ σ ≠ α do
15:             SEARCH(new_cursor[σ], e + 1, pos + 1)
16:     if e + 1 ≤ U[pos] then                          ▷ search for deletion
17:         for ∀σ ∈ Σ̂ do
18:             SEARCH(new_cursor[σ], e + 1, pos)
19:     if L[pos] ≤ e + 1 ≤ U[pos] then                 ▷ search for insertion
20:         SEARCH(cursor, e + 1, pos + 1)
```

Algorithm 5 Decides if next steps is left or right extension step

```
1:  function EXTEND(cursor, σ, pos)
2:      if pos = 0 ∨ Π[pos − 1] < Π[pos] then
3:          return extend_right(σ, cursor, index)
4:      return extend_left(σ, cursor, index)
```

Algorithm 6 Adjustments of algorithm 4 for Hamming distance

```
3: if pos = len(q) then                                   ▷ report condition
4:     if ~~ℒ[pos − 1] ≤ e ≤ 𝒰[pos − 1]~~ then            ▷ covered by condition in line 7
5:     REPORT(cursor)
6:     return
7: if 𝒰[pos] < e ∨ e < ℒ[pos] then return                 ▷ adjusted condition
   ...
16: ~~<deletion and insertion code>~~                     ▷ deleting lines 16-20
```

Also, using the search schemes $\mathcal{L}^{(ham)}$ and $\mathcal{U}^{(ham)}$ as in definition 22, reduces the number of possible recursive calls.

Optimizing Cursor Expansion A second way to optimize the algorithm is to add conditions to the cursor expansion. The cursor should only be expanded for all characters if another error is allowed. Otherwise, it is only necessary to expand the cursor for the character that is next in the search pattern P.

The lines 9 and 10 in algorithm 4 can be surrounded by an additional condition as described in algorithm 7. This adjustment works for Hamming distance and edit distance searches. It reduces unnecessary expansion of cursors. In practice, cursor expansions are especially costly because they require multiple lookups in the FM-index.

Algorithm 7 Replacement of line 9-10 in algorithm 4

```
9: if e + 1 ≤ 𝒰[pos] then          ▷ compute cursor extended by one character
10:     for ∀σ ∈ Σ̂ do
11:         new_cursor[σ] ← EXTEND(σ, cursor, index)
12: else
13:     new_cursor[α] ← EXTEND(α, cursor, index)
```

More Optimizations There are more optimization opportunities. It is possible to put stricter constraints on the search avoiding neighboring insertions and deletions that are only separated by substitutions. Introducing these kinds of constraints will change the set of results.

Furthermore, it is possible to add more restrictions besides the lower and upper bound. This can be used to avoid errors at the beginning or end of a query all together matching the properties of some modern sequence machines.

The downsides of this type of optimizations are that they are more sophisticated and harder to implement correctly. Comparing them to the seqan2 or seqan3 implementation is not feasible since such optimizations will not give the same set of results. Chapter 6 will take a look at such optimizations.

3.4 Benchmark

Tables 1 and 2 show a comparison between the seqan3 library and the pseudocode implementation. The main implementation differences between both implementations are that seqan3 works on π, L and U while the pseudocode uses the expanded versions Π, $\mathcal{L}$ and $\mathcal{U}$. The comparison in Table 1 shows an example closer to a practical application. It uses a randomly generated text and search patterns that are reproducing reads as they would be produced by an Illumina machine. The table shows that both implementations have similar running times.

Table 2 shows an example with a random text and random search patterns. The overall runtimes are much lower compared to the numbers from the previous test. This results from much quicker aborting searches and a much lower number of results. It also shows a clear difference between the seqan3 library and the pseudocode implementation.

The following sections of this thesis will include benchmarks based on the first test setup where the search queries are generated from the text.

	seqan3		pseudocode	
errors	Hamming	Edit	Hamming	Edit
0	17.1ns	17.3ns	17.8ns	17.5ns
1	54.4ns	62.0ns	53.8ns	68.8ns
2	80.1ns	348.6ns	83.1ns	358.3ns
3	101.0ns	3493.3ns	103.3ns	3358.9ns

Table 1: The table shows a comparison of runtimes of the seqan3 library against the pseudocode implementation. Using $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Text T is randomly uniformly distributed over Σ. The runtimes are averaged over $100'000$ simulated illumina reads generated with "mason". The Pseudocode and seqan3 implementation have similar runtimes. The differences are neglectable and both implementation can be seen as equal.

	seqan3		pseudocode	
errors	Hamming	Edit	Hamming	Edit
0	9.0ns	9.1ns	14.1ns	14.2ns
1	21.3ns	21.0ns	31.0ns	31.4ns
2	32.7ns	32.2ns	51.2ns	51.4ns
3	50.2ns	50.3ns	76.8ns	77.5ns

Table 2: The table shows a comparison of runtimes of the seqan3 library against the pseudocode implementation. Using $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Text T is randomly uniformly distributed over Σ. The runtimes are averaged over $1'000'000$ reads generated randomly. The Pseudocode and seqan3 implementation have distinct runtimes.

4 Extended Metric

The node count introduced in 2.5.2 has a few shortcomings when analyzing the performance of search schemes. The work from Pockrandt et al. [17] and Kianfar et al. [8] uses the node count for finding optimum search schemes using an integer linear program (ILP) or a mixed integer program (MIP). The original work of Kucherov et al. [10] uses a different formulation of the node count. It incorporates the probability of a string existing in the FM-index.

In this section, I want to show the importance of probability when estimating performance using node count. This approach will estimate the expected number of traversed nodes hence called *expected node count*.

4.1 Expected Node Count

Figure 6 shows a concrete search scheme for Hamming distance with 1 error with a query length of 4. Typical applications have queries with length hundreds or thousands of characters. Increasing the allowed distance k or using larger alphabets increases the amount of branching. If we look at an unbounded search, meaning unlimited errors are allowed, we can estimate the nodes in each level with $nodes_{l,\sigma} = \sigma^l$ where l is the level of the search scheme tree.

Even on small alphabets where $|\Sigma| = 4$ and a query length of $l = 20$ a node count of $nodes_{20,4} \approx 1099.5 * 10^9$ is expected. Considering a genome with one billion (10^9) base pairs (human genomes has around 3 billion base pairs [20]) our unbounded search would discover in each leaf about $\frac{1}{1099.5}$ of the results. From the pigeonhole principle, we can estimate that in the worst case every 1099.5 leaf will actually find a matching string. Since the pseudocode 4 on line 2 aborts early, not all nodes will be traversed. From this, we can conclude that a *cutoff level* exists at which search tree depth it is guaranteed that not all nodes will be traversed. This level depends on the length of a given text T.

$$\text{cutoff\_level}_{|\Sigma|}(|T|) = \lceil \log_{|\Sigma|} |T| \rceil$$

From this, even larger text and genome collections are expected to not traverse all nodes on unbound searches. Table 3 shows cutoff levels for certain text lengths assuming an alphabet of size $|\Sigma| = 4$. In the following this property is utilized to extend the *node count* to an *expected node count* that improves its usage as a performance indicator.

text size		cutoff level
Megabyte	10^6	10
Gigabyte	10^9	15
Terabyte	10^{12}	20
Petabyte	10^{15}	25

Table 3: Last level of a search scheme where all nodes are being traveresed. After the cutoff level not all nodes will be visited even by an unbound search.

4.2 Improved Performance Analysis

Instead of using the *node count* as a performance indicator for a given search scheme it is possible to incorporate the expectation of a node being traversed by the algorithm 4.

Taking the node count from definition 23 and applying the expectation of nodes being expanded the expected node count can be defined. Let $N = |T|$ be the size of the text and $\sigma = |\Sigma|$ the size of the alphabet.

Definition 27 (Expected node count - Hamming distance)

$$nc_{ham}(|P|) = \sum_{d=\mathcal{L}_{|P|-1}}^{\mathcal{U}_{|P|-1}} n_{|P|-1,d}$$

$$n_{l,d} = \begin{cases} n_{l-1,d} + (\sigma - 1) \cdot n_{l-1,d-1} & 1 \leq l \leq \lceil \log_\sigma N \rceil \wedge \mathcal{L}_l \leq d \leq \mathcal{U}_l \\ (n_{l-1,d} + (\sigma - 1) \cdot n_{l-1,d-1}) \cdot \frac{N}{\sigma^l} & l > \lceil \log_\sigma N \rceil \wedge \mathcal{L}_l \leq d \leq \mathcal{U}_l \\ 1 & l = 0 \wedge d = 0 \\ 0 & otherwise \end{cases}$$

The formula is similar to definition 23. The term $\frac{N}{\sigma^l}$ is added which has the property of always being below 1 since it is only applied when $l > \lceil \log_\sigma N \rceil$. It reflects the probability of a node existing assuming that all characters in the text are uniformly distributed. If the node query size is smaller than the given cutoff size as seen in Table 3, the definition 23 and 27 give the same results.

4.3 Benchmark

Having the *expected node count* defined, I want to compare it to the *node count*, showing that it is a more accurate performance indicator. For comparison, a setup is defined in which the alphabet is $\Sigma = \{A, C, G, T\}$, and a random text with length $|T| = 16 \cdot 10^6$ is

used.

Table 4 compares *node count* and *expected node count* against the actual running time. It uses the optimum search schemes given in seqan3, the backtracking expressed as a search scheme as described in Section 2.5 and compares these for allowed errors 0-3. The runtimes are measured using the pseudocode implementation.

	node count		expected node count		runtime in seconds	
Error - k	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{bt,k}$
0	200	200	14.024	14.024	0.5s	0.5s
1	40'799	80'600	28.048	433.320	1.8s	4.3s
2	11'518'400	21'413'400	42.072	7'643.820	2.8s	9.4s
3	2'325'700'000	4'245'310'000	56.096	93'115.600	4.8s	335.6s

Table 4: Shows data of optimum search schemes and search schemes based on backtracking. It compares the *node count*, *expected node count* and measured performance for Hamming distance searches. Used $|T| = 16'000'000$ and $|P| = 200$. The running times are the total over $100'000$ queries.

Using the standard node count as a performance indicator suggest that $\mathcal{S}_{OSS4}$ and $\mathcal{S}_{bt,k}$ have similar runtimes. It suggests that $\mathcal{S}_{OSS4}$ is roughly $2\times$ faster than $\mathcal{S}_{bt,k}$. Looking at the runtimes it is clear that this is not the case. Especially looking at $k = 3$ we can see that $\mathcal{S}_{bt,k}$ is more than $50\times$ slower than $\mathcal{S}_{OSS4}$. Comparing the measured timings with the *expected node count* shows a much closer correlation.

Only limited numbers of optimum search schemes are known. From these few data points, it is not possible to make any definite conclusion. But it gives a hint that *expected node count* is a better indicator.

5 Search Scheme Construction

The drawback of optimum search schemes is that these are only known for a small number k of allowed mismatches. In this section, I show that the ideas behind other approximate string searches can be used to build search schemes. I will provide instructions on how to convert these ideas into search schemes.

5.1 Backtracking

An example of backtracking converted to a search scheme was already given in Section 2.5. It keeps the search pattern in one piece and allowing it to be found with 0-k errors.

Definition 28 ($\mathcal{S}_{bt,k}$)

$$\begin{aligned}\pi_0 &= (0)\\ L_0 &= (0)\\ U_0 &= (k)\\ \mathcal{S}_{bt,k} &= ((\pi_0, L_0, U_0))\end{aligned}$$

5.2 Pigeonhole

In this section, I take the discussed pigeonhole principle from Section 2.4.2 and convert it into a search scheme. I construct two search schemes $\mathcal{S}_{ph}$ and $\mathcal{S}_{ph-opt}$.

Both search schemes follow closely the description of the pigeonhole principle. A search pattern is divided into $k+1$ pieces. Each piece will be the starting point of a search, leading to $k+1$ searches. Each search starts with 0 allowed errors.

In the first case, I construct a search scheme $\mathcal{S}_{ph,k}$ defined as $\mathcal{S}_{ph,k} = (S_0, ..., S_k)$ where $S_i = (\pi_i, L_i, U_i)$. The i-th search starts by searching for chunk i with exactly 0 errors. Afterward, all chunks to the left and then to the right are being searched with the allowed maximum number of k errors.

Definition 29 ($\mathcal{S}_{ph,k}$)

$$\begin{aligned}
&& \overset{\text{i-th chunk}}{} && \overbrace{\quad\quad}^{\text{chunks to the left}} && \overbrace{\quad\quad}^{\text{chunk to the right}}\\
\pi_i &= && \hat{i}, && i-1, ..., 0, && i+1, ..., k\\
L_i &= && 0, && 0, ..., 0, && 0, ..., 0\\
U_i &= && 0, && k, ..., k, && k, ..., k\\
\mathcal{S}_{ph,k} &= && ((\pi_0, L_0, U_0), ..., (\pi_k, L_k, U_k))
\end{aligned}$$

An example for $k = 2$ is given.

$$\begin{aligned}\mathcal{S}_{ph,2} = ((&012, 000, 022),\\ &(102, 000, 022),\\ &(210, 000, 022))\end{aligned}$$

This example shows an improvement opportunity. The search S_0 finds all matches with 0 errors including the 0-th chunk. This can be used to tighten the limitation to create a second search scheme by forcing the 0-th chunk of the other searches to have at least one error. Furthermore, a stricter upper limit can be chosen. This allows defining $\mathcal{S}_{ph-opt,k}$ with stricter boundaries.

Definition 30 ($\mathcal{S}_{ph-opt,k}$)

$$\begin{array}{rccc} & \text{i-th chunk} & \overbrace{\text{chunks to the left}} & \overbrace{\text{chunk to the right}} \\ \pi_i = & \widehat{i}, & \overbrace{i-1, ..., 0,} & \overbrace{i+1, ..., k} \\ L_i = & 0, & 1, ..., i, & i, ..., i \\ U_i = & 0, & \max(k, k-i+1), k-1, ..., k, & k, ..., k \\ \mathcal{S}_{ph-opt,k} = & & ((\pi_0, L_0, U_0), ..., (\pi_k, L_k, U_k)) & \end{array}$$

An example for $k = 2$ is given.

$$\begin{aligned}\mathcal{S}_{ph-opt,2} = ((&012, 000, 022),\\ &(102, 011, 022),\\ &(210, 012, 012))\end{aligned}$$

5.3 01*0 Seeds

Similar to pigeonhole, I use the 01*0 principle from Section 2.4.3 to construct search schemes. The 01*0 principle splits the search query into $k + 2$ chunks. All except the last chunk can be a starting point of the 01*0 criterion. Every chunk after the starting chunk can be an ending point. The search scheme can than be defined as $\mathcal{S}_{01*0} = (S_{0,0}, ..., S_{0,k}, S_{1,0}, ..., S_{1,k-1}, ..., S_{k,0})$ where each search $S_{i,j}$ is defined as $i \in \mathbb{N}_0$ be the starting chunk of the 01*0 criterion and $j \in \mathbb{N}_0$ be the number of chunks with 1 error.

Definition 31 ($\mathcal{S}_{01*0}$)

$$\begin{array}{lllll} \pi_{i,j} = & \overset{0}{\widehat{i}}, & \overset{1*}{\overbrace{...}}, & \overset{0}{\overbrace{i+j+1}}, & \overset{\text{left over chunks}}{\overbrace{i+j+1,...,k,i-1,...0}} \\ L_{i,j} = & 0, & 1,...,j, & j, & j \\ U_{i,j} = & 0, & 1,...,j, & j, & k,...,k \end{array}$$

$$\begin{aligned} \mathcal{S}_{01*0} = ((&\pi_{0,0},L_{0,0},U_{0,0}),...,(\pi_{0,k}, \quad L_{0,k}, \quad U_{0,k}), \\ &(\pi_{1,0},L_{1,0},U_{1,0}),...,(\pi_{1,k-1},L_{1,k-1},U_{1,k-1}) \\ &... \\ &(\pi_{k,0},L_{k,0},U_{k,0})) \end{aligned}$$

Example for $k = 2$ is given.

$$\begin{aligned} S_{01*0,2} &= ((0123,0000,0022),(0123,0100,0112),(0123,0122,0122) \\ &=(1230,0022,0022),(1230,0112,0112) \\ &=(2310,0022,0022)) \end{aligned}$$

The example shows an easy optimization opportunity. Since $\pi_{i,0}$ up to $\pi_{i,j}$ have the same order of pieces, it is possible to group them together. This leads to an optimized search scheme $\mathcal{S}_{01*0-opt,k}$.

Definition 32 ($\mathcal{S}_{01*0-opt,k}$)

$$\begin{aligned} \pi_{opt,i} &= \pi_{i,0} \\ L_{opt,i} &= \min(L_{i,j}) \\ U_{opt,i} &= \max(U_{i,j}) \\ \mathcal{S}_{01*0-opt,k} &= ((\pi_{opt,0},L_{opt,0},U_{opt,0}),...,(\pi_{opt,k},L_{opt,k},U_{opt,k})) \end{aligned}$$

The optimized example for $k = 2$ is given.

$$\begin{aligned} S_{01*0-opt,2} = ((&0123,0000,0122), \\ &(1230,0012,0122), \\ &(2310,0022,0022)) \end{aligned}$$

5.4 Suffix Filter

The construction of search schemes based on suffix filters follows similar principles as the previous constructions.

A search pattern is split into $k + 1$ chunks. These chunks can be divided into two

groups. The first group has to comply with the strong matching criterion representing the seeding phase. The second group of chunks resembles the extension phase.

Definition 33 ($\mathcal{S}_{sf,k}$)

$$\begin{aligned} \pi_i = &\quad \overbrace{i,...,k}^{\text{strong match}}, & \overbrace{i-1,...,0}^{\text{extension}} \\ L_i = &\quad 0,...,0, & 1 \\ U_i = &\quad 0,...,k-i, & k \\ \mathcal{S}_{sf,k} = &\quad ((\pi_0, L_0, U_0),...,(\pi_k, L_k, U_k)) \end{aligned}$$

Example for $k = 2$ is given.

$$\begin{aligned} \mathcal{S}_{sf,2} = ((&012, 000, 012), \\ &(120, 001, 012), \\ &(210, 011, 022)) \end{aligned}$$

5.5 Heuristic $H2$

Besides using the pigeonhole, 01*0, or suffix filter strategies to derive search schemes, I also take a look at constructing them from known optimum search schemes. The optimum search schemes are known for $k = 2$ and $k = 3$ which have $k + 2$ pieces. The optimum search scheme for $k = 1$ has 2 pieces. In this section, I take the optimum search scheme for $k = 3$ called $\mathcal{S}_{OSS4}$ apart and use it to implement a heuristic that allows creating search schemes that work for any k. The following dissecting method of $\mathcal{S}_{OSS4}$ for $k = 3$ also works for $k = 1$ and $k = 2$. The main idea is to use underlying patterns of the known optimum search schemes to recreate these and scale them for higher errors. When working on these heuristics, I have created over 7 different heuristics called *H1* to *H7*. But only two produced valid search schemes of that only one had a feasible running time. They are referred to as *heuristics*, because I hope that they are good construction methods, but I could not prove it from their intrinsic properties. In the following I will describe the second heuristic called *H2* denoted as $\mathcal{S}_{H2,k} = ((\pi_{H2,0}, L_{H2,0}, U_{H2,0}), (\pi_{H2,1}, L_{H2,1}, U_{H2,1}), ...)$.

I am going to take the search scheme $\mathcal{S}_{OSS4}$ that is described by Pockrandt [17]. When transforming it into a different representation, it is possible to gain more insights into the internal structure of the optimum search scheme.

Definition 34 (Optimum search scheme $\mathcal{S}_{OSS4}$)

$$\begin{aligned}\mathcal{S}_{OSS4} = ((&01234, 00003, 02233),\\ &(12340, 00022, 01223),\\ &(23410, 00111, 01123),\\ &(43210, 00000, 00333))\end{aligned}$$

Notice that the last search is searching for the chunks (43210) but it is possible to instead search for (34210) instead. Since the chunk 3 and 4 have both a lower and an upper boundary of 0 these two chunks can be searched in reversed order. The searches are equivalent to each other and do not change the *node count*. For easier discussion I rewrite $\mathcal{S}_{OSS4}$ as 3 matrices M_π, M_L and M_U grouping each search as a row.

$$M_\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 1 & 0 \\ 3 & 4 & 2 & 1 & 0 \end{pmatrix} \quad M_L = \begin{pmatrix} 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad M_U = \begin{pmatrix} 0 & 2 & 2 & 3 & 3 \\ 0 & 1 & 2 & 2 & 3 \\ 0 & 1 & 1 & 2 & 3 \\ 0 & 0 & 3 & 3 & 3 \end{pmatrix}$$

To recreate these patterns the configuration space has to be decided. In the upcoming approach, I decided to give a construction plan for a heuristic that takes as an input the number of allowed mismatches $k \in \mathbb{N}_0$ and the number of pieces $p \in \mathbb{N}_0$ where $k \leq p$. The number of searches in the search scheme is fixed to $k + 1$.

Looking at all M_π a clear pattern emerges. We can see how the i-th row starts with i and goes up to $k + 1$ and then it continues going from $i - 1$ to 0. This is used as the definition to order the chunks $\pi_{H2,i}$.

Definition 35 ($\pi_{H2,i}$)

$$\pi_{H2,i} = (\overbrace{i, ..., p-1}^{\text{extending right}}, \overbrace{i-1, ..., 0}^{\text{extending left}})$$

The lower bounds which are shown in M_L also have an apparent structure. By

marking all zeros in the upper left corner with 'x' it gets obvious.

$$M_{L,x} = \begin{pmatrix} \mathtt{x} & \mathtt{x} & \mathtt{x} & \mathtt{x} & 3 \\ \mathtt{x} & \mathtt{x} & \mathtt{x} & 2 & 2 \\ \mathtt{x} & \mathtt{x} & 1 & 1 & 1 \\ \mathtt{x} & 0 & 0 & 0 & 0 \end{pmatrix}$$

This can be used to formulate a function that generates this pattern directly for any given k.

Definition 36 ($L_{H2,i}$)

$$L_{H2,i} = (\overbrace{0, ..., 0}^{\times(p-i-1)}, \overbrace{k-i, ..., k-i}^{\times(i+1)})$$

The pattern in M_U is not that obvious. To extract it we need to perform multiple steps. First, we compute a modified $M_{L\text{»}1}$ which shifts the columns of M_L by one to the right. The new entries are filled with zeros and the last column is removed.

$$M_L = \begin{pmatrix} 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\Rightarrow M_{L\text{»}1} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

As a next step, we take the upper boundaries M_U and subtract $M_{L\text{»}1}$ from it, computing M'_U. As the last step, the entries in each row of the matrix of M'_U have to be reordered in each row. The reordering has to correspond as they are ranked by the

entries of M_π.

$$M'_U = M_U - M_L = \begin{pmatrix} 0 & 2 & 2 & 3 & 3 \\ 0 & 1 & 2 & 2 & 3 \\ 0 & 1 & 1 & 2 & 3 \\ 0 & 0 & 3 & 3 & 3 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 2 & 2 & 3 & 3 \\ 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 3 & 3 & 3 \end{pmatrix}$$

$$\Rightarrow M'_{U,ord} = \begin{pmatrix} 0 & 2 & 2 & 3 & 3 \\ 1 & 0 & 1 & 2 & 2 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 3 & 3 & 0 & 0 \end{pmatrix}$$

Looking at $M'_{U,ord}$, 4 properties can be identified. First, it can be seen that each column has all values between 0 and 3. Second, the upper left part of the matrix has zeros on its diagonal. Next, the last row has 3 times the entry 3 followed by the rest of the row filled with 0. Finally, the row-wise neighboring numbers increase with the distance to the 0 entries in each column.

This allows us to add some descriptions to matrix $M'_{U,ord}$ which is known to have $k = 3$ and $p = 5$. The height of the matrix is determined by the number of searches $k + 1 = 4$. The width has to follow the number of pieces which is $p = 5$.

From this, it is possible to give a general description on how to construct an arbitrary upper bound U by reconstructing $M'_{U,ord}$ only depending on input k and p.

First, an intermediate matrix M is constructed, reflecting the properties of $M'_{U,ord}$.

$$M = \left(\begin{array}{c|c} M_k & M_t \\ \hline M_b & \end{array}\right) = \left(\begin{array}{cccc|ccc} m_{0,0} & m_{0,1} & \dots & m_{0,k-1} & m_{0,k} & \dots & m_{0,p-1} \\ m_{1,0} & m_{1,1} & \dots & m_{1,k-1} & m_{1,k} & \dots & m_{1,p-1} \\ \dots & & & \dots & \dots & & \dots \\ m_{k-1,0} & m_{k-1,1} & \dots & m_{k-1,k-1} & m_{k,k} & \dots & m_{k,p-1} \\ \hline m_{k,0} & m_{k,1} & \dots & m_{k,k-1} & & & \end{array}\right)$$

The matrix M is split into three sub matrices called M_k, M_b and M_t. Each of these

matrices have to satisfy several characteristics.

$$\begin{aligned} \forall\ 0 \leq i < k \rightarrow & \qquad m_{j,i} = (j + i \mod k) \\ \forall\ 0 \leq i < k \rightarrow & \qquad m_{k,i} = k \\ \forall\ i \geq k \wedge 0 \leq j \leq k \rightarrow & \qquad m_{j,i} = k - j \end{aligned}$$

The formula above ensures that each column in M_k has all numbers between 0 and $(k-1)$ and maintains zeros on the diagonal. The second formula ensures that all entries of M_b have the value k. The last formula fills the columns of M_t with decreasing numbers. On top of this, it has to fulfill the property of increasing numbers on the horizontal when moving away from the zeros. Specified as a formula:

$$\begin{aligned} \forall m_{i,i} = 0 \wedge j < i \rightarrow m_{i,j+1} > m_{i,j} \\ \forall m_{i,i} = 0 \wedge j > i \rightarrow m_{i,j-1} < m_{i,j} \end{aligned}$$

This property might not be fulfilled for a row j. In this case, the column i of the first violating element $m_{j,i}$ has to be adjusted. The elements $m_{0,i}$ until $m_{k-1,i}$ without the element $m_{i,i}$ will be permuted until none of the element in this column violate this property. In a second step, the matrix M will be reordered to M_{ord} according to the π entries. This is the reverse step of transforming M'_U to $M'_{U,ord}$.

$$M_{ord} = \begin{pmatrix} m_{0,\pi_{H2,0,0}} & m_{0,\pi_{H2,0,1}} & ... & m_{0,\pi_{H2,0,p-1}} \\ m_{1,\pi_{H2,1,0}} & m_{1,\pi_{H2,1,1}} & ... & m_{1,\pi_{H2,1,p-1}} \\ ... & & ... & ... \\ m_{k,\pi_{H2,k,0}} & m_{k,\pi_{H2,k,1}} & ... & m_{k,\pi_{H2,k,p-1}} \end{pmatrix}$$

The matrix M_{ord} can be used to define $U_{H2,i}$ and thus define $\mathcal{S}_{H2,k,p}$.

Definition 37 ($U_{H2,i}$)

$$U_{H2,i} = (0 + m_{ord,i,0}, L_{H2,i,0} + m_{ord,i,1}, ..., L_{H2,i,p-2} + m_{ord,i,p-1})$$

Definition 38 ($\mathcal{S}_{H2,k,p}$)

$$\mathcal{S}_{H2,k,p} = ((\pi_{H2,0}, L_{H2,0}, U_{H2,0}), ..., (\pi_{H2,k}, L_{H2,k}, U_{H2,k}))$$

5.6 Benchmark

The test setup involves a text T over $\Sigma = \{A, C, G, T\}$ where the length of $|T| = 1'000'000'000$. The text is randomly generated. The search queries have a length of 100 and reproduce reads with errors that are typical for Illumina machines. Table 5 shows the *node count* of the previous defined heuristics. It shows that for $k = 1$, the search scheme $\mathcal{S}_{H2,1,2}$ has the same node count as search scheme $\mathcal{S}_{OSS4}$ and that for $k \in \{0, 2, 3\}$ search scheme $\mathcal{S}_{H2,k,k+2}$ has the same number as the optimum search schemes. From this can be derived that search schemes based on $H2$ reconstruct optimum search schemes for $k \in \{0, 1, 2, 3\}$. The table also shows that $H2$ based search schemes have the lowest node count for every number of k compared to the other heuristics. As discussed in Chapter

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$
0	10^1	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**
1	10^3	15.25	**7.85**	8.62	**7.85**	**7.85**	**7.85**	8.62
2	10^5	15.15	**8.28**	11.57	10.22	8.67	8.67	**8.28**
3	10^7	11.18	**6.25**	11.70	9.23	7.42	6.65	**6.25**
4	10^9	6.53		8.52	6.29	5.00	3.99	**3.72**
5	10^{11}	3.14		5.00	3.41	2.79	1.98	**1.83**
6	10^{12}	12.85		23.70	15.44	12.91	8.24	**7.59**
7	10^{14}	4.55		9.66	5.98	5.09	2.96	**2.74**
8	10^{15}	14.16		33.57	19.80	16.81	9.32	**8.55**
9	10^{17}	3.92		10.34	5.88	4.87	2.56	**2.43**

Table 5: Showing the node count of different search schemes. Assuming $|P| = 100$, $|\Sigma| = 4$ and Hamming distance The lowest node count is highlighted. Search schemes based on heuristic $H2$ have the lowest node count accross all allowed mismatches. All numbers are shown in Table 11.

4, the node count suggests similar runtimes between the search schemes. This is a wrong assumption. Table 6 shows the *expected node count*. It predicts that the running times of all search schemes with the exemption of $\mathcal{S}_{bt,k}$ are similar which is confirmed by tables 7 and 8.

Table 12 and Table 13 show a complete overview of *expected node count* and running times. The data from Table 12 shows that for $k \in \{3, 4, 5, 6\}$ the best performing search scheme is $\mathcal{S}_{ph-opt,k}$ and for $k \in \{7, 8, 9, 10\}$ the best performing search scheme is $\mathcal{S}_{01*0-opt,k}$ assuming Hamming distance is used. Table 16 and 17 show similar results for patterns $|P| = 250$ and $|P| = 50$. It seems that the best search scheme performance

is based on the amount of errors in percentage of the length of the pattern.

errors	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+2}$
0	**16.2**	**16.2**	**16.2**	**16.2**	**16.2**	**16.2**
1	434.6	**32.5**	**32.5**	**32.5**	**32.5**	**32.5**
2	6'762.3	**48.7**	**48.7**	**48.7**	**48.7**	**48.7**
3	73'618.6	**65.0**	**65.0**	**65.0**	**65.0**	**65.0**
4	597'693.4	-	81.8	**81.2**	**81.2**	82.3

Table 6: The table show a subview of Table 12. It shows the *expected node count* with $|T| = 1'000'000'000$, $|\Sigma| = 4$ and $|P| = 100$. It predicts that a search based on backtracking is magnitudes slower than the others and that the others have similar running times. Smallest node count are highlighted.

Runtimes - Hamming distance

errors	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+2}$
0	**17.5**	**17.5**	17.6	17.8	17.6	17.7
1	159.7	62.2	**55.0**	62.4	61.9	55.2
2	1793.1	90.5	82.7	83.3	88.5	**80.7**
3	17327.4	104.8	105.4	**93.6**	105.2	97.6
4	160737.1	-	130.3	**101.5**	120.8	115.5

Table 7: Comparision of runtimes based on different search schemes using the pseudocode implementation. Times given in microseconds. Used Hamming distance, $T = 1'000'000'000$, $|\Sigma| = 4$ and $|P| = 100$. Extended view is shown in Table 13.

Runtimes - Edit distance

errors	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{s,f,k}$	$\mathcal{S}_{H2,k,k+2}$
0	16.5	**16.3**	19.5	19.3	19.3	19.2
1	291.4	67.8	**60.8**	69.3	67.5	64.4
2	8103.2	363.3	480.7	506.6	371.0	**343.5**
3	240236.2	3152.7	4964.9	4274.1	**3066.7**	3267.3
4	5106618.3	-	77660.6	57792.0	38243.1	**31158.9**

Table 8: Comparision of runtimes based on different search schemes using the pseudocode implementation. Times reported in microseconds. Used edit distance, $T = 1'000'000'000$, $|\Sigma| = 4$ and $|P| = 100$. Extended view is shown in Table 14.

6 Optimizations

6.1 Edit Distance Enhancement

This section will look at optimizations for edit distance searches by reducing the complexity of the search tree. Let assume $S_1 = ACGT$ and $S_2 = CCGT$. There exists multiple edit transcripts that transform S_1 into S_2. The pseudocode implementation already ensures for Hamming distance the smallest edit transcript is applied. This follows from the fact that there is only the *substitution* (S_σ) operation which is applied to one character and each character can only have one operation applied. For edit distance this limitation is not enough. There are the operations *deletion* (D) and *insertion* (I_σ) where *deletion* is applied to a character but *insertion* is applied in front or behind a character allowing multiple operation for one chunk. Some combinations of operations like an *insertion* followed by a *deletion* can be combined into a *substitution* operation. Avoiding a subset of operation combinations that are also covered by a smaller set of operations reduces the number of paths in the suffix tree which yields in an improved runtime.

To formalize this reduction, the edit transcript has to be rewritten to reflect the internal workings of the search scheme algorithm. When running algorithm 4, the expression Π_{pos} indicates which character to process next. The operations *substitution* and *insertion* are directly affecting the character Π_{pos}. The operation *deletion* is performed directly in front or behind the character Π_{pos}. The actual position is determined if it is a forward or a backward step.

To simplify this issue Π is split into Π_{fwd} and Π_{bwd}. Π_{fwd} includes all entries of $\Pi_i \geq \Pi_0$ and Π_{bwd} all entries of $\Pi_i \leq \Pi_0$.

Following the search algorithm and looking at steps involving only entries of Π_{fwd} it is possible to rewrite the operations in each step as a list. In addition to the 3 known operations, this list also uses the operation *match* (M) to indicate that no transformation is applied to a character. To transform S_1 to S_2 and assuming that $\Pi_{fwd} = (0, 1, 2, 3)$ there are multiple possible operation lists of which some are $(S_C MMM)$, $(DI_C MMM)$, $(I_C DMMM)$ and $(S_C MMMI_A)$.

In the following, I show how to reduce certain operations to a lower number of operations and how some operation combinations are covered by other operation combinations.

Lemma 2 (Deletion followed by insertion (DI))
A deletion operation followed by an insertion can be replaced by a substitution operation.

$$\begin{array}{llll} ACGT & A\ \ CGT & \ \ ACGT & ACGT \\ S\ |\ |\ | & D\,I\ |\ |\ | & I\,D\ |\ |\ | & S\ |\ |\ |\ I \\ CCGT & \ \ CCGT & C\ \ CGT & CCGT\,A \end{array}$$

Figure 11: Shows possible operation lists of algorithm 4 that transform $S_1 = ACGT$ to $S_2 = CCGT$.

Proof. A (DI) transformation has a distance count of 2. This can be replaced with a substitution (S) with distance count 1. Since $2 > 1$, an (S) is always possible when (DI) transformation is possible. □

Lemma 3 (Deletion followed by any number of substitutions followed by an insertion (DS*I))
A deletion followed by an arbitrary number of substitutions followed by an insertion can be replaced by only substitutions.

Proof. A single substitution (S) can be split into one insertion and one deletion (ID). Applying this to (DS*I) transformation gives (D (ID)* I). In case there are zero substitution operations, one substitution is possible because of lemma 2. Otherwise, we can expand the transformation by pulling out one substitution to (DI (DI)* DI). This can again be collapsed into (S S* S) or just (S*) which has one operation less than the original (DS*I) which makes this transformation always possible. □

Proofs for (ID) and (IS*D) follow the same argument as proof for lemma 2 and lemma 3.

Lemma 4 (Substitution followed by deletion (SD))
A substitution operation followed by a deletion can be replaced by a deletion followed by a substitution.

Proof. A substitution followed by a deletion (SD) can be expanded to (DID) that is equivalent to (DS). (SD) and (DS) have the same error count and can always be performed. □

Lemma 5 (Substitution followed by insertion (SI))
A substitution followed by an insertion can be replaced by an insertion followed by a substitution.

Proof. A substitution followed by an insertion (SI) can be expanded to (DII) which is equivalent to (IS). (SI) and (IS) have the same error count and can always be performed. □

From the previous lemmas, it is possible to conclude some optimizations. The search code 4 only has a condition for error limits when accessing the recursive calls for insertions and deletions. It is possible to add some extra conditions.

The insertion recursion must only be called if the previous recursion call was a *match* or an *insertion*.

The same is valid for recursion calls for deletion operation that should only be called if the previous call was a *match* or a *deletion*. Calls to substitution or matches are always possible. The improved pseudocode is referred to as *enhanced code*.

Benchmark

Table 9 shows running times of searches based on edit distance. Search schemes $\mathcal{S}_{sf,k}$ and $\mathcal{S}_{H2,k,k+2}$ are measured. The table shows clearly the performance advantage of the *enhanced code*. While small errors as $k = 2$ have an 3× improvement, higher error rates as $k = 6$ show over 1000× improvement of runtimes.

errors	scale	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+2}$
		pseudocode		enhanced code	
0	10^{-5}	1.93	1.92	**1.78**	1.80
1	10^{-5}	6.75	6.44	5.00	**4.70**
2	10^{-5}	37.10	34.35	11.52	**10.27**
3	10^{-5}	306.67	326.73	**33.73**	34.41
4	10^{-5}	3824.31	3115.89	119.91	**102.21**
5	10^{-4}	3675.13	3052.15	48.00	**36.70**
6	10^{-4}	56308.96	57277.20	183.17	**130.77**
7	10^{-4}			783.32	**541.20**
8	10^{-4}			4155.68	**2590.14**
9	10^{-3}			**1353.54**	1643.23

Table 9: Comparision of pseudocode to enhanced code. Edit based searches. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted. Tables 15 and 14 show a complete overview with more search schemes.

6.2 Query Constraints

Until now the algorithms assume that errors can occur anywhere on the search pattern. Modern sequence machines have certain error patterns in their reads. Some machines produce a low rate of errors at the beginning of a read. In this section, I develop an additional constraint to the existing lower and upper bound of a search scheme which limits the appearance of errors in partial sections of the query.

To achieve this, I group characters into $g \in \mathbb{N}_0$ groups. Every character has to be part of exactly one group. Let P be the search query then $0 < g \leq |P|$. Let $G \in \mathbb{N}_0^g$ be a g-tuple which specify the maximum numbers of allowed mismatches per group. The function $proj : \mathbb{N}_0 \to \mathbb{N}_0$ is a surjection which projects the i-th character of a query into to the j-th group.

Example Example for a query of length 9 where only the last 6 characters are allowed to have up to 3 errors and the first 3 characters have to be error-free.

$$proj = (0, 0, 0, 1, 1, 1, 1, 1, 1)$$
$$G = (0, 3)$$

Benchmark

In this particular implementation, I choose to use a *proj* function that evenly distributes the characters into 3 groups while keeping neighboring characters together. With this $G = (x_0 x_1 x_2)$ groups each third of characters of a pattern P together. The implementation is referred to as *constraint code* and is build upon the *enhanced code* of the previous Section 6.1.

Table 10 shows the benchmark comparing four different scenarios with $G \in \{(kkk), (0kk), (0k0), (000)\}$. It shows a clear increase in performance by constraining the queries. Since the number of results changes with the constraints, the usefulness depends on the prevailing circumstances. If such characteristics of a search pattern is known, this can provide a huge performance gain.

errors	enhanced code	constraint code			
		kkk	$0kk$	$0k0$	000
0	16.1μs	16.0μs	16.1μs	16.1μs	16.1μs
1	49.5μs	49.8μs	42.9μs	36.3μs	28.4μs
2	109.6μs	110.5μs	86.8μs	63.9μs	35.6μs
3	340.0μs	342.4μs	220.8μs	124.3μs	43.6μs
4	1113.2μs	1115.6μs	620.7μs	273.2μs	49.1μs
5	3977.5μs	3916.8μs	1795.5μs	603.9μs	59.6μs
6	14042.7μs	14032.4μs	5351.9μs	1450.7μs	63.9μs
7	55539.6μs	55581.6μs	17575.6μs	5581.6μs	65.7μs
8	288135.2μs	289236.1μs	48865.0μs	13638.4μs	73.3μs
9	1133573.5μs	1139464.2μs	173718.2μs	76037.1μs	76.2μs

Table 10: Comparing running times of the *enhanced code* of the *constraint code*. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Generated $100'000$ patterns that simulate "illumina' reads. Using $\mathcal{S}_{H2,k,k+2}$ search schemes.

7 Discussion

A major disadvantage of optimum search schemes is that they are only known for 3 or fewer mismatches. The motivation of this thesis is to make search scheme based approximate string searches feasible for larger numbers of mismatches.

The implementation from Chapter 3 provides the basis to evaluate this research question. I show that this implementation performs similarly to the algorithm in the *seqan3* library. The simplified structure of my algorithm allows it to be analyzed thoroughly and to enable further optimizations.

Previous work theorized the metric of *node count* to assess the performance of search schemes. In Chapter 4 this metric is refined to a metric named *expected node count*. It considers the likelihood of a string appearing in a reference text. Measuring running times of different search schemes show that the *expected node count* is a much closer and more accurate performance indicator.

Chapter 5 presents methods to construct search schemes. Known approximate string matching algorithms (backtracking, pigeonhole principle, 01*0 seeds, and suffix filter) are used to generate search schemes. In addition, I identify underlying patterns in known optimum search schemes. Based on these patterns I build a generator named $H2$ which produces new search schemes. Comparison of generated search schemes for different numbers of allowed mismatches show that search schemes based on $H2$, pigeonhole, and 01*0 seeds perform best. The results show that depending on the number of allowed mismatches and the distance metric (Hamming distance or edit distance) a different search scheme generator should be picked.

Chapter 6 introduces two optimizations. The first optimization combines redundant edit transcript operations to reduce the paths through the FM-index. In practice, this reduces the running time for edit distance based searches significantly.

The second part looks at additional constraints applied to a search query. It formalizes constraints for combined parts of the query, limiting only these parts to lower numbers of mismatches. This thesis shows that limiting the first third of the query to 0 mismatches decreases the running times considerably.

The initial goal of this thesis was to find search schemes allowing for a higher number of mismatches. Search schemes based on 01*0 seeds, pigeonhole and $H2$ achieve this for Hamming distance and edit distance based searches.

Search schemes based on $H2$ are optimal for 0-3 mismatches. This thesis did not prove that $H2$ based search schemes are also optimum search schemes for a larger number

of mismatches.

It is important to point out that the pseudocode implementation in this thesis has a few differences to the seqan3 library. The seqan3 library implementation allows specifying different numbers for allowed substitutions, insertions, and deletions.

A similar extension could be introduced to the presented pseudocode. It might be worth using the concept of *cost* instead of *error* or *mismatches*. Further development could also allow *wildcards* as they are often seen in genomic data processing. Using a scoring matrix that specifies the cost of one operation makes the integration of *wildcards* easy. A scoring matrix also enables having separate alphabets for a text T and a search pattern P.

A wide range of additional optimizations is imaginable. It is possible to integrate a caching mechanism since the first steps in every search are applied to the same characters, but also building an index over multiple search queries might be a feasible improvement.

All in all, **The Search** for faster search algorithms has made considerable advancements in the last decade but stays an important and fundamental challenge of bioinformatics.

References

[1] Michael Burrows and David J Wheeler. "A block-sorting lossless data compression algorithm". In: (1994).

[2] Haoyu Cheng, Ming Wu, and Yun Xu. "FMtree: A fast locating algorithm of FM-indexes for genomic data". In: *Bioinformatics* 34.3 (2018), pp. 416–424.

[3] Paolo Ferragina and Giovanni Manzini. "An experimental study of a compressed index". In: *Information Sciences* 135.1-2 (2001), pp. 13–28.

[4] Paolo Ferragina and Giovanni Manzini. "Indexing compressed text". In: *Journal of the ACM (JACM)* 52.4 (2005), pp. 552–581.

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

[6] Richard W Hamming. "Error detecting and error correcting codes". In: *The Bell system technical journal* 29.2 (1950), pp. 147–160.

[7] Juha Kärkkäinen and Joong Chae Na. "Faster filters for approximate string matching". In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 84–90.

[8] Kiavash Kianfar et al. "Optimum Search Schemes for Approximate String Matching Using Bidirectional FM-Index". In: *arXiv preprint arXiv:1711.02035* (2017).

[9] Donald E. Knuth. *The art of computer programming III*. 2nd ed. Addison-Wesley Professional, 1998. ISBN: 0201896850,9780201896855.

[10] Gregory Kucherov, Kamil Salikhov, and Dekel Tsur. "Approximate string matching using a bidirectional index". In: *Theoretical Computer Science* 638 (2016), pp. 145–158.

[11] Tak Wah Lam et al. "High throughput short read alignment via bi-directional BWT". In: *2009 IEEE International Conference on Bioinformatics and Biomedicine*. IEEE. 2009, pp. 31–36.

[12] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[13] Veli Mäkinen. "Compact suffix array". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2000, pp. 305–319.

[14] Udi Manber and Gene Myers. "Suffix arrays: a new method for on-line string searches". In: *siam Journal on Computing* 22.5 (1993), pp. 935–948.

[15] Edward M McCreight. "A space-economical suffix tree construction algorithm". In: *Journal of the ACM (JACM)* 23.2 (1976), pp. 262–272.

[16] Christopher Pockrandt, Marcel Ehrhardt, and Knut Reinert. "EPR-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional fm indices". In: *International Conference on Research in Computational Molecular Biology*. Springer. 2017, pp. 190–206.

[17] Christopher Maximilian Pockrandt. "Approximate String Matching: Improving Data Structures and Algorithms". PhD thesis. 2019.

[18] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. "Bidirectional search in a string with wavelet trees". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2010, pp. 40–50.

[19] Esko Ukkonen. "Approximate string-matching over suffix trees". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1993, pp. 228–242.

[20] J Craig Venter et al. "The sequence of the human genome". In: *science* 291.5507 (2001), pp. 1304–1351.

[21] Christophe Vroland et al. "Approximate search of short patterns with high error rates using the 01 0 lossless seeds". In: *Journal of Discrete Algorithms* 37 (2016), pp. 3–16.

[22] Laurence OW Wilson et al. "VARSCOT: variant-aware detection and scoring enables sensitive and personalized off-target detection for CRISPR-Cas9". In: *BMC Biotechnology* 19.1 (2019), p. 40.

[23] Sun Wu and Udi Manber. "Fast text searching allowing errors". In: *Communications of the ACM* 35.10 (1992), pp. 83–92.

Appendix

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k-1}$	$\mathcal{S}_{H2,k,k-2}$	$\mathcal{S}_{H2,k,k-3}$
0	10^1	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**
1	10^3	15.25	**7.85**	8.68	8.62	7.85	**7.85**	**7.85**	**7.85**	8.62	9.72
2	10^5	15.15	**8.28**	9.46	11.57	13.54	10.22	8.67	8.67	**8.28**	8.84
3	10^7	11.18	**6.25**	8.70	11.70	14.11	9.23	7.42	6.65	**6.25**	6.49
4	10^9	6.53		6.28	8.52	10.61	6.29	5.00	3.99	**3.72**	3.83
5	10^{11}	3.14		3.72	5.00	6.23	3.41	2.79	1.98	**1.83**	1.86
6	10^{12}	12.85		17.71	23.70	30.00	15.44	12.91	8.24	**7.59**	7.75
7	10^{14}	4.55		7.25	9.66	12.20	5.98	5.09	2.96	**2.74**	2.78
8	10^{15}	14.16		25.11	33.57	42.80	19.80	16.81	9.32	**8.55**	8.74
9	10^{17}	3.92		7.81	10.34	13.18	5.88	4.87	2.56	**2.43**	2.44
10	10^{18}	9.78		21.38	28.13	36.14	15.35	13.40	6.58	**6.10**	6.10
15	10^{25}	2.45		7.40	9.83	13.00	4.76	4.71	1.72	**1.59**	1.60
20	10^{30}	9.83		38.90	51.44	67.18	23.36	22.90	7.22	**6.73**	6.78
30	10^{40}	2.30		12.47	16.83	21.95	6.07	7.89	1.82	**1.75**	1.75
40	10^{47}	5.29		37.08	49.35	62.87	17.63	28.49	4.46	**4.29**	**4.29**

Table 11: Showing the node count of different search schemes. Assuming $|P| = 100$, $|\Sigma| = 4$ and Hamming distance The lowest node count is highlighted. Search schemes based on heuristic $H2$ have the lowest node count accross all allowed mismatches.

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^1	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**	**1.62**
1	10^1	43.46	**3.25**	4.87	**3.25**	**3.25**	**3.25**	**3.25**	**3.25**	**3.25**	**3.25**
2	10^1	676.23	**4.87**	9.75	**4.87**	**4.87**	**4.87**	**4.87**	**4.87**	**4.87**	**4.87**
3	10^1	7361.86	**6.50**	16.24	**6.50**	**6.50**	**6.50**	**6.50**	**6.50**	**6.50**	6.56
4	10^1	59769.34		24.52	8.18	**8.13**	**8.12**	**8.12**	**8.12**	8.23	11.20
5	10^2	37573.51		4.19	1.30	1.07	**1.02**	**1.02**	1.03	1.52	4.37
6	10^2	187477.29		8.91	2.53	4.38	2.89	**2.14**	2.62	6.41	30.02
7	10^2	755515.56		18.01	**4.46**	48.08	19.06	12.15	18.57	51.02	195.44
8	10^2	2491695.94		31.13	**6.79**	318.72	152.58	43.41	116.68	304.55	635.99
9	10^3	680108.91		4.68	**1.00**	159.97	79.77	16.74	59.89	102.13	144.94
10	10^3	1553877.79		6.69	**2.11**	683.58	309.41	67.54	217.32	223.95	595.24
15	10^5	127649.89		**1.49**	6.98	790.88	305.84	60.93	189.56	257.08	257.18
20	10^6	26098.32		**7.15**	72.93	1230.02	406.24	141.13	356.96	228.53	294.64
30	10^8	527.65		**5.19**	40.53	236.98	135.60	11.00	67.38	73.15	90.61
40	10^{10}	7.94		1.42	6.01	14.40	8.12	**1.05**	3.60	3.45	3.81
50	10^{10}	10.61		7.64	21.15	39.12	25.13	**6.14**	16.60	14.60	14.10
60	10^{11}	**1.33**		7.24	8.98	10.75	4.03	7.48	2.85	2.79	2.83
70	10^{11}	**1.58**		16.57	18.22	19.75	4.03	16.89	5.27	5.47	5.71
80	10^{11}	**1.71**		26.30	27.53	28.55	3.36	26.59	8.86	9.27	9.69
90	10^{11}	**1.71**		35.09	35.69	36.11	2.42	35.21	13.87	14.46	15.06
100	10^{11}	**1.71**		45.43	44.57	44.57	1.73	44.00	**1.71**	**1.71**	**1.71**

Table 12: Showing the expected node count of different search schemes. Assuming $|P| = 100$, $|\Sigma| = 4$, $|T| = 1'000'000'000$ and Hamming distance. The lowest expected node count is highlighted. Depending on the allowed mismatches $\mathcal{S}_{sf,k}$, $\mathcal{S}_{01*0-opt}$ or $\mathcal{S}_{01*0}$ has the lowest node count.

errors	scale	$\mathcal{S}_{bk,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^{-5}	**1.75**	1.75	1.76	1.76	1.76	1.78	1.76	1.76	1.77	1.76
1	10^{-5}	15.97	6.22	5.96	5.50	6.23	6.24	6.19	6.24	5.52	**5.40**
2	10^{-5}	179.31	9.05	9.44	8.27	9.76	8.33	8.85	9.11	8.07	**7.98**
3	10^{-5}	1732.74	10.48	13.03	10.54	11.61	**9.36**	10.52	10.28	9.76	9.62
4	10^{-5}	16073.71		16.81	13.03	14.00	**10.15**	12.08	11.87	11.55	12.44
5	10^{-4}	11035.39		2.16	1.60	1.65	**1.07**	1.35	1.31	1.46	2.48
6	10^{-4}			2.98	2.15	3.46	**1.71**	1.85	2.01	3.28	10.54
7	10^{-4}			4.36	**2.97**	20.59	7.32	5.75	8.29	17.59	62.99
8	10^{-4}			6.05	**3.82**	122.16	54.82	17.62	45.10	106.17	205.46
9	10^{-3}			0.82	**0.53**	60.62	28.91	6.40	22.44	32.51	46.61
10	10^{-3}			1.11	**0.98**	252.46	107.54	24.17	72.60	74.87	183.66
15	10^{-1}										
20	10^{0}										
30	10^{2}										
40	10^{4}										

p

Table 13: Runtimes of searches based on pseudocode using Hamming distance. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted.

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^{-5}	1.65	**1.63**	1.79	1.95	1.95	1.93	1.93	1.91	1.92	1.91
1	10^{-5}	29.14	6.78	6.25	6.08	6.90	6.93	6.75	6.84	6.44	**6.06**
2	10^{-5}	810.32	36.33	49.31	48.07	78.40	50.66	37.10	36.78	**34.35**	37.41
3	10^{-5}	24023.62	315.27	473.80	496.49	797.10	427.41	**306.67**	327.91	326.73	330.95
4	10^{-5}	510661.83		7534.05	7766.06	13025.41	5779.20	3824.31	3326.18	**3115.89**	3196.26
5	10^{-4}			10798.96	11853.55	16908.33	5752.50	3675.13	3266.73	3052.15	**3036.93**
6	10^{-4}							**56308.96**	61285.87	57277.20	57165.66
7	10^{-4}										
8	10^{-4}										
9	10^{-3}										
10	10^{-3}										
15	10^{-1}										
20	10^{0}										
30	10^{2}										
40	10^{4}										

Table 14: Runtimes of searches based on *pseudocode* using edit distance. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted.

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^{-5}	1.78	1.79	1.79	1.80	1.79	1.79	**1.78**	1.80	1.80	1.79
1	10^{-5}	24.59	5.17	4.92	**4.60**	5.05	5.00	5.00	5.08	4.70	4.82
2	10^{-5}	567.15	10.77	13.00	12.79	16.66	13.00	11.52	11.74	10.27	**10.24**
3	10^{-5}	11963.91	**32.71**	50.00	48.75	69.32	40.57	33.73	34.19	34.41	33.08
4	10^{-5}	214592.72		224.34	223.51	321.92	149.82	119.91	107.03	**102.21**	104.71
5	10^{-4}			100.23	104.16	152.18	56.96	48.00	37.52	**36.70**	40.67
6	10^{-4}			447.33	482.22	606.04	223.98	183.17	133.94	**130.77**	162.99
7	10^{-4}			1682.58	1870.85	2477.21	738.61	783.32	**511.88**	541.20	968.94
8	10^{-4}			8849.77	9339.36	16154.10	3896.25	4155.68	**1959.17**	2590.14	4464.23
9	10^{-3}						1991.17	1353.54	**1341.05**	1643.23	1590.17
10	10^{-3}										
15	10^{-1}										
20	10^{0}										
30	10^{2}										
40	10^{4}										

Table 15: Runtimes of searches based on *enhanced code* using edit distance. Hamming based searches. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 100$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted.

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^{-5}	3.11	**3.10**	3.11	3.11	3.11	3.12	3.11	3.11	3.11	3.11
1	10^{-5}	17.41	**10.36**	13.00	10.86	10.38	10.44	10.38	10.45	10.66	10.88
2	10^{-5}	150.85	18.41	27.06	20.59	20.32	**15.39**	17.66	17.42	18.41	18.93
3	10^{-5}	1439.05	25.18	43.79	31.25	30.99	**18.09**	24.78	23.97	25.55	26.61
4	10^{-5}	12704.22		62.10	41.95	41.68	**19.55**	31.71	29.69	32.32	33.49
5	10^{-4}	8480.94		7.82	5.30	5.29	**2.08**	3.86	3.55	3.80	3.98
6	10^{-4}			9.60	6.32	6.29	**2.15**	4.51	4.04	4.36	4.61
7	10^{-4}			11.75	7.42	7.40	**2.21**	5.28	4.65	5.00	5.22
8	10^{-4}			13.55	8.48	8.44	**2.26**	5.85	5.24	5.51	5.73
9	10^{-3}			1.53	0.98	0.96	**0.23**	0.65	0.57	0.60	0.62
10	10^{-3}			1.73	1.09	1.08	**0.23**	0.71	0.61	0.65	0.68
15	10^{-1}										
20	10^{0}										
30	10^{2}										
40	10^{4}										

Table 16: Runtimes of searches based on *pseudocode* using Hamming distance. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 250$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted.

errors	scale	$\mathcal{S}_{bt,k}$	$\mathcal{S}_{OSS4}$	$\mathcal{S}_{01*0,k}$	$\mathcal{S}_{01*0-opt,k}$	$\mathcal{S}_{ph,k}$	$\mathcal{S}_{ph-opt,k}$	$\mathcal{S}_{sf,k}$	$\mathcal{S}_{H2,k,k+1}$	$\mathcal{S}_{H2,k,k+2}$	$\mathcal{S}_{H2,k,k+3}$
0	10^{-5}	**0.87**	0.88	0.97	0.92	0.92	0.92	0.91	0.95	0.94	0.91
1	10^{-5}	11.60	**2.86**	3.53	2.87	3.03	2.97	2.99	2.99	3.15	3.53
2	10^{-5}	150.11	6.28	8.65	6.23	5.34	**3.94**	4.68	4.51	5.61	8.17
3	10^{-5}	1504.94	18.03	19.72	**11.73**	27.45	13.28	14.39	15.48	19.15	28.45
4	10^{-5}	13076.01		38.33	**23.67**	442.55	219.21	100.26	141.39	94.23	159.15
5	10^{-4}	9357.36		**10.50**	15.22	478.40	142.38	99.12	124.66	60.22	96.91
6	10^{-4}			**34.66**	156.18	2886.42	1149.12	495.61	683.27	321.48	569.75
7	10^{-4}			**162.67**	994.18	16716.13	5460.24	2577.53	3877.95	2079.17	3747.99
8	10^{-4}			**498.18**	4316.88			11954.60	15591.69	9346.93	9280.89
9	10^{-3}			**294.56**	2312.22						
10	10^{-3}			**831.28**							
15	10^{-1}										
20	10^{0}										
30	10^{2}										
40	10^{4}										

Table 17: Runtimes of searches based on *pseudocode* using Hamming distance. Hamming based searches. Used random text $|T| = 1'000'000'000$, $|\Sigma| = 4$, $|P| = 50$. Generated $100'000$ patterns that simulate "illumina' reads. Results are given in average runtime per pattern. Fastest runtime is highlighted.