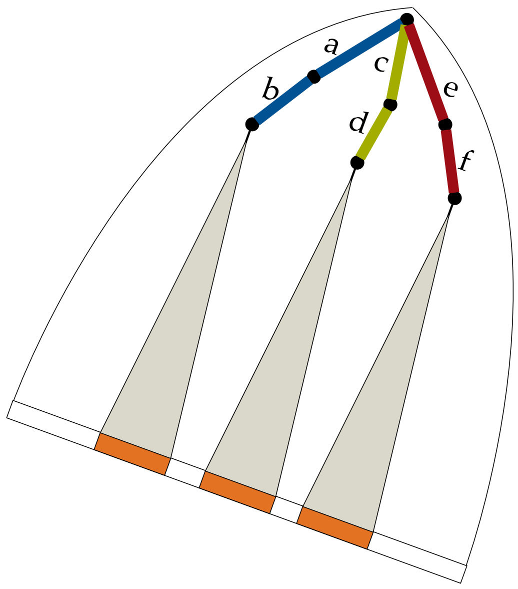
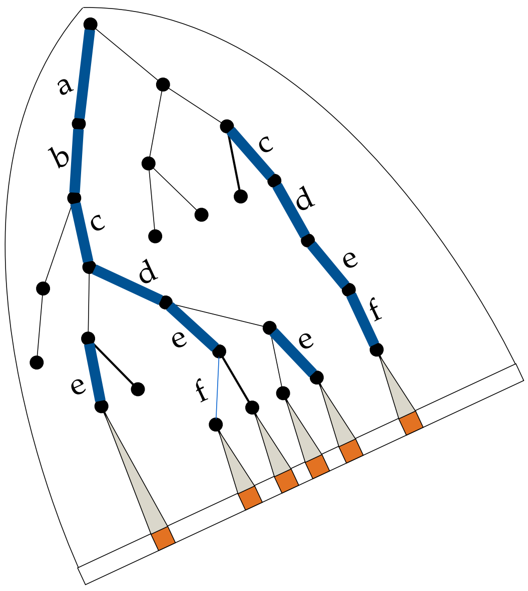
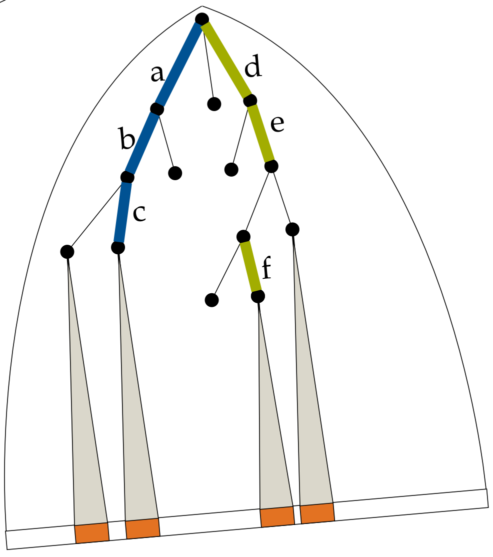
Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen

Approximate Pattern Matching with Index Structures

Johannes Krugel







Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen

Approximate Pattern Matching with Index Structures

Johannes A. Krugel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Ernst W. Mayr
2. Univ.-Prof. Dr. Stefan Kramer, Johannes Gutenberg-Universität Mainz

Die Dissertation wurde am 06.05.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.01.2016 angenommen.

Zusammenfassung

Ziel dieser Arbeit ist es, einen Überblick über das praktische Verhalten von Indexstrukturen und Algorithmen zur approximativen Textsuche (approximate pattern matching, APM) zu geben, abhängig von den Eigenschaften der Eingabe. APM ist die Suche nach Zeichenfolgen in Texten oder biologischen Sequenzen unter Berücksichtigung von Fehlern (wie z. B. Rechtschreibfehlern oder genetischen Mutationen). In der Offline-Variante dieses Problems kann der Text vorverarbeitet werden, um eine Indexstruktur aufzubauen bevor die Suchanfragen beantwortet werden.

Diese Arbeit beschreibt und diskutiert praktisch relevante Indexstrukturen, Ähnlichkeitsmaße und Suchalgorithmen für APM. Wir schlagen einen neuen effizienten Suchalgorithmus für Suffixbäume im externen Speicher vor. Im Rahmen der Arbeit wurden mehrere Indexstrukturen und Algorithmen für APM implementiert und in einer Softwarebibliothek bereitgestellt; die Implementierungen sind effizient, stabil, generisch, getestet und online verfügbar.

Wir haben reale Testinstanzen zusammengestellt sowie Textgeneratoren zur Erzeugung künstlicher Testinstanzen mittels stochastischer Prozesse implementiert; die Parameter der stochastischen Prozesse können erlernt werden. Alle Testinstanzen wurden hinsichtlich ihrer statistischen Eigenschaften analysiert. Dies ermöglicht eine experimentelle Untersuchung unter realen und/ oder sehr kontrollierten Bedingungen.

In einer Reihe von Experimenten haben wir das Verhalten der einzelnen Indexstrukturen und Algorithmen in Abhängigkeit der Eigenschaften der Eingabe untersucht. Abschließend geben wir Empfehlungen für eine geeignete, situationsbezogene Auswahl der Verfahren und der Parameterwerte.

Abstract

The aim of this thesis is to provide an overview of the practical performance of index structures and algorithms for approximate pattern matching (APM) depending on the properties of the input. APM is the problem of searching a pattern in a text or biological sequence tolerating some errors (e. g. spelling mistakes or genetic mutations). In the offline variant, the text can be preprocessed to build an index structure before answering the search queries.

This thesis describes and discusses several practically relevant solutions for APM, including index structures, similarity measures, and search algorithms. We propose a new efficient algorithm for APM of multiple patterns using suffix forests in external memory. As part of this project, several index structures and algorithms were implemented in a software library; the implementations are efficient, stable, generic, tested, and available online.
We assembled real world test instances and implemented text generators to produce synthetic texts using stochastic processes; the parameters of the processes can be learned. All test instances were analyzed regarding their statistical properties. This makes it possible to perform experiments under realistic and/or very controlled conditions.

In a series of experiments, we investigated the behavior of the individual index structures and algorithms depending on the properties of the input. Then we change the perspective and give recommendations for choosing the respective best solution in different practical settings, and furthermore, we indicate appropriate parameter values.

Contents

List of Figures

List of Tables

Part I

Introduction

1 Introduction

Approximate pattern matching is a problem that occurs in many different real world scenarios.

1.1 Motivation

In many practical applications of computer science it is necessary to search in a text or in a collection of texts:

1. When **browsing a library**, the user is often interested in all books containing a given search term.
2. While **reading or writing a document**, the user might want to find the pages containing a term.
3. Performing a **lookup in a dictionary** requires finding all matches for a specified word [Boy11].
4. In **information retrieval**, the goal is to get all documents that match a possibly complex search query.
5. **Plagiarism detection** is based on finding segments in a collection of documents that are similar to parts of a given suspect document. [Pot+12]
6. **Database systems** support operations to search a record field for a specified text. [HL09]
7. **Record linkage** aims at identifying rows in a database that refer to the same real world entity [Win06].

These examples are usually concerned with natural language texts. There are, however, also numerous applications with other types of sequences, particularly biological sequences. The molecules of DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) consist of sequences of four chemical bases adenine, cytosine, guanine, and thymine (for DNA) or uracil (for RNA). They can therefore be represented by a sequence of characters over the alphabet ‘‘A’’, ‘‘C’’, ‘‘G’’, and ‘‘T’’ or ‘‘U’’, respectively. Proteins molecules consist of chains of amino acids (21 different in total for a human) and can therefore also be represented by sequences of characters. Many applications using such biological data require a search inside the sequences. Exemplary applications are:

8. **Genome sequencing** is usually based on combining short pieces of sequenced DNA (so called *reads*) to form one long genomic sequence. This can be done with the help of a *reference genome*; this method requires to locate the position of the small pieces inside the reference genome [SJ08] (this problem is also called *read mapping*).
9. **Sequence alignment** can be used to determine the functional or evolutionary similarity of genomic sequences (of the same or different individuals or species) by analyzing the sequences. It is often based on finding regions of high similarity of the involved sequences [SW81].
10. **Searching a DNA database** can be used to solve many different problems such as finding the origin of a protein inside a genomic sequence or examining a DNA database of criminals for a piece of DNA found at a crime scene [Gus97].

All those applications mentioned are – from a computer science point of view – very related. They are all based on *pattern matching* (also called *string matching*), where search patterns have to be found in a text or a set of strings. There are many more similar applications for natural language texts, biological sequences or even other types of sequences, such as structured texts (e. g., XML) or audio sequences.

Furthermore, all these applications have in common that there might be some kind of errors, differences, or variations in the sequences involved. For natural language texts such errors may result from:

- Spelling mistakes (e. g., "necessary" → "neccesary", "neighbor" → "neigbor")
- Typos (e. g., "necessary" → "necessray", "neighbor" → "neghbour")
- Inflections of words (e. g., "necessary" → "necessarily", "neighbor" → "neighbors")
- Alternative spellings (e. g., "neighbor" → "neighbour")
- Optical character recognition errors (e. g., "necessary" → "neccssarg", "neighbor" → "ndghboi")

Errors in biological sequences can originate, for example, from [Gus97]:

- Genetic variations among individuals or species, e. g., single-nucleotide polymorphisms (SNP)
- Other mutations (e. g., insertions of longer pieces into DNA sequences)
- Sequencing errors (e. g., because the biochemical processes of DNA sequencing cannot always unambiguously determine each base)

Practical applications have to solve the above mentioned problems even if the underlying data might have suffered from errors. The general goal is then the following:

Find in a text (or set of texts) the occurrences of a search pattern allowing some errors.

This problem is called *approximate pattern matching* and studied in this thesis.[1] (The case where no errors are allowed is for distinction called *exact pattern matching*.) The occurrences of a pattern are called *matches*. Approximate pattern matching problems are studied in many research areas including theoretical computer science, database systems, bioinformatics, natural language processing etc. There are several names for the same or very related problems:

- Approximate pattern matching [WM92a]
- Approximate string matching [Nav01]
- Approximate dictionary lookup [OT10]
- Approximate dictionary searching [Boy11]
- Pattern matching with errors [NR02]
- Text searching with errors [WM92b]
- Dictionary matching with errors [Col+04]
- Inexact matching [Gus97]
- *k*-error-matching [Cha+06a]
- *k*-differences matching [Gus97]
- Fuzzy search in strings [VL09]

[1]The term "approximate" means here to also find deviating matches, and therefore *more* matches than in the exact case. It is *not* used as in "approximation algorithm" where the algorithm might be allowed to return *less* results or suboptimal solutions [Nav01].

The applications differ in several dimensions:

- Underlying problem instance: text, dictionary, collection of texts, . . .
- Type of the search queries: existence query, position query, top-K-query, . . .
- Similarity or distance measure used for the strings
- Size of the problem instance: e. g., only a short hand-written text or gigabytes of data
- Time constraints to answer the queries
- And many more dimensions

In the following sections we describe and formalize the problem definition and different types of problem instances, as well as search queries for exact and approximate search.

1.2 Basics and notation

In this thesis we use standard notation for mathematics, alphabets, and strings which we extended by some hopefully intuitive own notation for string operations. All used notation is summarized in the following paragraphs. Lower case characters are used to denote numbers, characters, and strings, while upper case characters denote sets, data structures, and algorithms.

Mathematics.

$\mathbb{N}$	Natural numbers including 0
$\mathbb{R}$	Real numbers, $\mathbb{R}^+$ for positive and $\mathbb{R}_0^+$ for non-negative real numbers
$\mathcal{P}$	Power set
P	Probability of an event
log	Logarithm (with base 2 if not indicated otherwise)

Alphabet.

Σ	denotes an *alphabet*, which is simply a finite set, e. g., $\Sigma = \{0, 1\}$ or $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ or $\Sigma = \{$ all 94 ASCII characters $\}$.
σ	denotes the *size of the alphabet*, $\sigma := \lvert\Sigma\rvert$.
u, w	$\in \Sigma$ are the variable names usually used for the elements of an alphabet, the so-called *characters*.
$\$ \notin \Sigma$	is a special character used to mark the end of a string for rather technical reasons in some of the algorithms.
$\Sigma_\$$	$= \Sigma \cup \{\$\}$ is called the *extended alphabet*.
$<$	is a total ordering defined on the characters of an alphabet, e. g., the usual order of characters in the English alphabet. The special symbol \$ is considered smaller than any other character. Having such a total ordering is needed for some of the solutions discussed in this thesis.

Strings. A *string* (also called *character sequence* or sometimes simply *text*) over an alphabet Σ is a (possibly empty) ordered sequence of characters. A string of length $a \in \mathbb{N}$ can formally be defined as a-tuple $(u_1, \ldots, u_a) \in \Sigma^a$.

“. . .” denotes string constants, e. g., “ACGA” = (A, C, G, A).

ϵ denotes the *empty string* and is defined as $\epsilon :=$ “” $\in \Sigma^0$.

Σ^* denotes the *set of all finite strings* over an alphabet Σ, formally: $\Sigma^* := \bigcup_{i \in \mathbb{N}, i \geq 0} \Sigma^i$.

Σ^+ denotes the *set of all non-empty finite strings* over an alphabet Σ, formally $\Sigma^+ := \bigcup_{i \in \mathbb{N}, i \geq 1} \Sigma^i = \Sigma^* \setminus \{\epsilon\}$.

$p, r, s, t \in \Sigma^*$ are the variable names usually used for strings throughout this thesis.

String operations. For strings $r = (u_1, \ldots, u_a)$ and $s = (w_1, \ldots, w_b)$ we introduce the following operations.

$|r|$ denotes the *length* of the string r, $|r| := a$, e. g., |“ACGC”| = 4.

$r_{[i]}$ denotes the *character at position* i of the string r (counting from 1), formally $r_{[i]} := u_i$, e. g., “ACGC”$_{[3]}$ = G

$r_{[i\,..\,j]}$ denotes the *substring* of r from position i to position j (including), formally: $r_{[i\,..\,j]} := (u_i, \ldots, u_j)$. For $j < i$ it is defined to be the empty string ϵ. For the ease of presentation it is also referred to as *infix* $[i, j]$. (In the literature, is also called *infix*, *factor*, or *subsequence*.)

$r_{[..\,j]}$ denotes the *prefix* of r ending at position j (having length j), formally: $r_{[..\,j]} := r_{[1\,..\,j]}$. It is also referred to as *prefix* j.

$r_{[i\,..]}$ denotes the *suffix* of r starting at position i (having length $|r| - i + 1$), formally: $r_{[i\,..]} := r_{[i\,..\,|r|]}$. It is also referred to as *suffix* i.

$r \circ s$ is the *concatenation* of the two strings r and s, formally: $r \circ s := (u_1, \ldots, u_a, w_1, \ldots, w_b)$, e. g., “ACGC” $\circ$ “T” = “ACGCT”. It can more compactly also be written simply as $r\,s$. The concatenation is defined analogously for single characters.

$\overleftarrow{r}$ denotes the *reversed string* r, formally: $\overleftarrow{r} := (u_a, \ldots, u_1)$, e. g., $\overleftarrow{\text{“ACGC”}}$ = “CGCA”.

lcp : $\Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ is a function that computes the length of the *longest common prefix* of two strings, e. g., lcp(“**AC**GC”, “**AC**TC”) = 2.

Lexicographic ordering.

$\prec\ \subset \Sigma^* \times \Sigma^*$ is the lexicographical ordering of strings derived from the ordering $<$ of characters of the alphabet, e. g., “ACGC” $\prec$ “ACTC” for the usual English ordering of characters. The empty string ϵ is considered smaller than any other string.

$\prec\ \subset \mathbb{N} \times \mathbb{N}$ is the lexicographical ordering of suffix positions of a string r (the string r will be clear from the context). For $i, j \in \mathbb{N}$ it is formally defined as $i \prec j :\Leftrightarrow r_{[i\,..]} \prec r_{[j\,..]}$.

String measures.

δ : $\Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ is a function that maps pairs of strings to a real value representing their *distance*.

ϕ : $\Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ is a function that maps pairs of strings to a real value representing their *similarity*.

$k \in \mathbb{R}$ is the variable used in this thesis to refer to values of distance and similarity.

1.3 Problem instances

The applications discussed in Section 1.1 differ in which kind of data the search is carried out. The user is either interested in searching in one long string, in a dictionary of short strings, or in a collection of long strings. These three types of problem instances are formalized now and we discuss how the types of problems can be converted into each other.

1.3.1 Text

Probably the most basic case of pattern matching is searching in one long string, which is called the *text* in this thesis (even if it does not consist of natural language but, for example, represents biological data). The text is denoted by the variable $t \in \Sigma^*$ and its length is denoted by $n := |t|$. Popular applications are searching in a document (Application 2) and genome sequencing (Application 8). Many solutions for pattern matching are designed for this rather simple type of problem instance.

In some applications, the text is structured as words, especially in natural language texts [FF07] (by defining special word delimiter characters such as spaces or punctuation marks). In this case, the user might only be interested in matches that start at the beginning of a word – and not in matches that begin in the middle of a word or that span multiple words. We do, however, not investigate this special case in more detail here.

1.3.2 Dictionary

In other applications a search has to be carried out in some kind of *dictionary*[2] [BYN98]. The dictionary (also called, e. g., *lexicon* [ZD95]) is a set of usually rather short strings and denoted by the variable D. It can formally be defined as set of strings $D = \{ s_1, s_2, \ldots, s_l \in \Sigma^* \} \in \mathcal{P}(\Sigma^*)$ with $l := |D|$.[3] The elements of the set are called *dictionary entries* and we are interested in finding those dictionary entries that match a query string in their entirety. Practical applications of such problem instances are the dictionary lookup of natural language words (Application 3) and searching a database for records where a field matches a specified string (Application 6).

1.3.3 Collection of texts

Instead of only searching in *one* text, it can be desirable to simultaneously search in *several* texts at once. Such a problem instance is called *collection of strings*, denoted by the variable C and can formally be defined as set of strings $C = \{ t_1, t_2, \ldots, t_l \in \Sigma^* \} \in \mathcal{P}(\Sigma^*)$ with $l := |C|$. It is formally the same as a dictionary but with a different intention: here we refer to problem instances where we are interested in matching substrings and not only matching entire entries as in the dictionary case. Practical examples are, for example, searching a library (Application 1), Information Retrieval (Application 4), and searching a DNA database for DNA fragments (Application 10).

1.3.4 Convertibility

Some solutions for approximate pattern matching problems are specialized for one particular type of problem instance (most commonly for one text or a dictionary). However, it is often possible to transform the input to another form, making it feasible to use a solution for one type to also

[2]In computer science, the term *dictionary* is sometimes also used to refer to an associative array. In the context of pattern matching and in this thesis it refers to a set of strings.

[3]$\mathcal{P}$ denotes the power set.

solve problem instances of another type. Here we briefly discuss how these conversions can be achieved.

Text $\rightarrow$ dictionary. There are several possibilities to simulate searching a text t by using a solution to search in a dictionary.
A text t of length n can be seen as the set of all its $\mathcal{O}(n^2)$ substrings. Searching in the text can then be simulated by searching in the dictionary of all the substrings. However, this approach is not very practical for longer texts because of the deteriorated time and space bounds.
A text of length n can also be represented as the set of its $n+1$ suffixes. It is then possible to simulate a search in the text by adapting some existing solutions for searching in the dictionary of the suffixes. (This approach is, for example, used by suffix trees discussed in Section 2.2.2.)
(If the text is structured as words, we can build the dictionary containing all those words and perform the search in the dictionary. With this approach it is, however, not possible to also find substrings that do not start at a word boundary.)

Dictionary $\rightarrow$ text. Assume we have a solution to search in a text and want to use it to search in a dictionary D of alphabet Σ. This is possible by concatenating all dictionary entries $s_i \in D$ with a distinct delimiter symbol $\$_i \notin \Sigma$ appended to each entry (to mark the end of an entry) [Gus97][Section 6.4]. The resulting string is over the alphabet $\Sigma_\$:= \Sigma \cup \{ \$_1, \ldots, \$_l \}$ and of the form $t(D) := s_1 \$_1 s_2 \$_2 \ldots s_l \$_l$. In this context, the variable n denotes the length of the concatenation: $n := |t(D)| = \sum_{s_i \in D} |s_i \$_i| = l + \sum_{s \in D} |s|$.
Searching in the dictionary D can then be simulated by searching in the text $t(D)$. This approach might, however, yield matches that span multiple entries and should be filtered out. Additionally, it is possible to also find *substrings* of dictionary entries (and not only entire matching entries); if those matches are not desired in the application, they also have to be discarded.
This transformation yields a solution which is in some sense more powerful, because it allows to also find substrings of dictionary entries. In some applications, specialized solutions for dictionaries can be more appropriate in practice (see Chapter 2).

Collection of texts $\rightarrow$ text. Using a solution for searching in one text to search in a collection of texts C can be done very similarly. We can again concatenate all $t_i \in C$ with a distinct delimiter symbol $\$_i$ appended to each entry. The resulting string is of the form $t(C) := t_1 \$_1 t_2 \$_2 \ldots t_l \$_l$. The variable n again denotes the length of the concatenation $t(C)$. Unlike in the case above, we only have to discard matches that cover boundaries of the texts (marked with $\$_i$).
A collection of texts can thus be transformed into one text without remarkable negative effects on the time or space consumption [Gus97][Section 6.4]. This type of problem instance is therefore not discussed separately in greater detail in the following.

Text $\rightarrow$ collection of texts. This conversion is trivial by wrapping the one text in a set.

Collection of texts $\rightarrow$ dictionary. This conversion can be achieved transitively with the methods described above.

Dictionary $\rightarrow$ collection of texts. This conversion can also be achieved transitively.

1.4 Query types

In some applications we want to know *whether or not* a pattern occurs in the problem instance, in other applications we are interested in the *positions* of the matches. We therefore formalize different types of queries, first for exact and then generalized for approximate pattern matching.

1.4.1 Exact pattern matching

For exact pattern matching, we are given a problem instance (a text t or a dictionary D), and a search pattern p of length m. We are interested in the exact matches of the p in the problem instance.

Position query. If we are interested in the *positions* of the matches, a so-called *position query* (in the literature also called *enumerative*, *locating*, or *listing query*) has to be executed [MN05b; Gro11].
A position query in a given *text* t is a function $R_{\text{pos}} : \Sigma^* \to \mathcal{P}(\mathbb{N})$ that maps a search pattern p of length m to all positions in the text where an occurrence of p starts (some solutions use end positions instead). It is formally defined as follows:

$$R_{\text{pos}}(p) := \left\{ i \in [1, n] \;\middle|\; p = t_{[i\,..\,i+m]} \right\}$$

A position query in a given *dictionary* D is a function $R_{\text{pos}} : \Sigma^* \to \mathcal{P}(\mathbb{N})$ that maps a search pattern p to a set of identifiers $i \in [1, l]$. This result set can contain either 0 or 1 elements and is formally defined as follows (the definition is already oriented for compatibility with the corresponding approximate query):

$$R_{\text{pos}}(p) := \{ i \in [1, l] \mid p = s_i \in D \}$$

Counting query. In some applications it might be sufficient to only know *how often* a search pattern occurs in a problem instance and the specific positions are irrelevant. In this case a *counting query* can be used [MN05b; Gro11]. A counting query is a function $R_{\text{count}} : \Sigma^* \to \mathbb{N}$ that maps a search pattern p to the number of occurrences of p in the problem instance, formally:

$$R_{\text{count}}(p) := \left| R_{\text{pos}}(p) \right|$$

Boolean query. If is it sufficient to determine whether or not a search pattern occurs in a text, a *boolean query* (also called *existence query* or *decision query* [Abo+04; MN05b; Gro11]) can be used. It is a function $R_{\text{bool}} : \Sigma^* \to \mathbb{B}$ that maps a search pattern to a boolean value ($\mathbb{B} = \{ \text{true}, \text{false} \}$):

$$R_{\text{bool}}(p) := \begin{cases} \text{false} & \text{if } R_{\text{count}}(p) = 0 \\ \text{true} & \text{otherwise} \end{cases}$$

In some pattern matching solutions it is easier or faster to answer a counting or a boolean query compared to a position query, because these values can be computed directly without the need to explicitly enumerate the set of matching positions. This is discussed in more detail for the individual solutions.

1.4.2 Approximate pattern matching

Approximate pattern matching is based on some kind of measure for strings, i. e., either a distance measure $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}$ or a similarity measure $\phi : \Sigma^* \times \Sigma^* \to \mathbb{R}$ that maps pairs of strings to a real value representing their distance or similarity, respectively. When we want to refer generically to either a distance or similarity measure, we use the term *string measure*. Here we assume such a function is given together with a search tolerance $k \in \mathbb{R}$ (also called *score limit* for similarity measures).[4] The functions R_{pos}, R_{count}, and R_{bool} are overloaded with the additional parameter $k \in \mathbb{R}$ for the ease of presentation.

Several algorithms perform differently not only depending on the search tolerance k, but also on the relative amount of errors allowed in relation to the length m of the search pattern p. Therefore the so-called *error level* α is introduced and defined as follows: $\alpha := \frac{k}{m}$ [Nav01].

Position query. The approximate matches of a pattern p in a text t are all those substrings $t_{[i\,..\,j]}$ starting at position i and ending at position j that have a distance of at most k to the search pattern. Formally they are defined by the function $R_{\text{pos}} : \Sigma^* \times \mathbb{R} \to \mathcal{P}(\mathbb{N})$ (here for starting positions, but in some solutions the end positions are used instead):

$$R_{\text{pos}}(p, k) := \left\{\, i \in [1, n] \;\middle|\; \exists j : \delta(p, t_{[i\,..\,j]}) \le k \,\right\}$$

When performing approximate pattern matching in a dictionary D, the position query asks for the identifiers i of all dictionary entries s_i that have a distance of at most k to the search pattern. For *exact* pattern matching, this set includes 0 or 1 element, but for approximate pattern matching, up to all l elements can be contained. Formally, the result set is given by the function $R_{\text{count}} : \Sigma^* \times \mathbb{R} \to \mathbb{N}$:

$$R_{\text{pos}}(p, k) := \{\, i \in [1, l] \mid s_i \in D, \delta(p, s_i) \le k \,\}$$

For a similarity measure ϕ instead of a distance function δ, the definitions have to be changed so that all strings with $\delta(p, x) \ge k$ are included.
(*Exact* pattern matching can be seen as a special case with $k = 0$ if the distance function satisfies the *identity of indiscernibles*: $\forall_{x,y} : \delta(x, y) = 0 \Leftrightarrow x = y$.)

Counting query. A counting query for approximate pattern matching can be defined the same as for exact pattern matching as the number of matches. Formally it is a function $R_{\text{count}} : \Sigma^* \times \mathbb{R} \to \mathbb{N}$:

$$R_{\text{count}}(p, k) := \left|R_{\text{pos}}(p, k)\right|$$

Boolean query. A boolean query returns true or false, depending on whether or not the problem instance contains a match of the search pattern with distance at most k. Formally it is a function $R_{\text{bool}} : \Sigma^* \times \mathbb{R} \to \mathbb{B}$:

$$R_{\text{bool}}(p, k) := \begin{cases} \text{false} & \text{if } R_{\text{count}}(p, k) = 0 \\ \text{true} & \text{otherwise} \end{cases}$$

Top-K-query. For approximate pattern matching another type of query is sometimes used, namely the *top-K-query*[5] [VL09; Yan+10; Den+13; KS13]. The result contains up to K matches of the search pattern in the problem instance, in particular those with the lowest distance (or

[4] Different possibilities to define those functions together with their advantages and disadvantages are discussed in Section 3.1.

[5] The variable K is used in uppercase here because k is already used for the search tolerance.

highest similarity respectively). It can formally be defined as a function $R_{\text{top}} : \Sigma^* \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$ for a dictionary $D = \{ s_1, \ldots, s_l \}$ as follows:

$$\begin{aligned} R_{\text{top}}(p, K) &:= \text{first } K \text{ entries of } L, \text{ where} \\ &\qquad L := \text{sequence of identifiers } i \in [1, l], \text{ sorted by } \delta(p, s_i) \text{ in ascending order} \end{aligned}$$

The definition can analogously be extended for a text (instead of a dictionary) and a similarity function (instead of a distance function). The results are required to be sorted in many applications. We discuss top-K-queries only marginally in this thesis.

1.4.3 Online vs. offline pattern matching

Approximate pattern matching can be divided into two categories depending on whether the text is completely available before answering the search queries or not [Nav01]. For *online* approximate pattern matching, the text and the search pattern are given simultaneously, so that there is not time to preprocess the text beforehand. For *offline* approximate pattern matching (also called *indexed* approximate string matching), the text is given in advance, making it possible to build a data structure (also called *index*) on the text beforehand (this phase is called *preprocessing* or *index construction*). The index can then help to speed-up the search queries.
This situation is very similar to searching in a real book all the pages that treat a specific topic: *online* approximate pattern matching corresponds to flipping through all pages, while *offline* approximate pattern matching corresponds to looking up the relevant pages in an alphabetical index.
In this thesis we focus on the offline version and therefore on *approximate pattern matching with index structures*.

1.5 Contributions and structure of this thesis

In this thesis we give an overview of the different approaches for approximate pattern matching. We provide an accessible and unified presentation of the practically usable state-of-the-art solutions including index structures (Chapter 2) and algorithms for approximate search (Chapter 3). We include approaches from different fields of research, among others from theoretical computer science, database systems, and bioinformatics. Other presentations and comparisons often focus only on solutions from one field (a recent exception is [Wan+14]). We provide for each solution a description of the ideas, data structures and algorithms, as well as an analysis of the space consumption and running times, and furthermore a discussion of the advantages, disadvantages and possible extensions.
Since there is an enormous number of data structures and algorithms for pattern matching we selected the in our view most promising solutions for practical applications. We decided to only include solutions which are practically feasible also for bigger input (e. g., in the order of magnitude of the human genome[6]), because for small texts simpler online algorithms can be faster [Nav01]. We only consider index structures which are of general use and are powerful enough to efficiently solve also other problems (like, e. g., exact pattern matching[7]) as compared to indexes specialized for and restricted to approximate pattern matching. We also focus on algorithms that can be used with several index structures and are not restricted to just one index. This allows to combine different index structures with different algorithms and to choose them independently. We exclude solutions using speed-up heuristics that might miss some actual

[6]The human genome can be represented with about 3 billion characters.
[7]Exact pattern matching also often constitutes the basis for approximate pattern matching solutions [Ohl13].

matches or are specific for a concrete application (like, e. g., the famous BLAST algorithm [Alt+90] or heuristic algorithms used in web search engines).
We furthermore propose a new efficient algorithm for approximate pattern matching of multiple patterns using suffix forests stored in external memory.
We efficiently implemented several index structures and algorithms for approximate pattern matching, most implementations have been done during or are based on students' projects. We do not only provide experimental or prototypical implementions but the index structures and algorithms have a uniform interface, are reusable, tested, available online and are implemented in a software library (SeqAn, see below in Section 1.6). Much more work and care therefore has to be invested, but the resulting implementations also are of a higher value for others.[8]

We furthermore give an overview of several other software libraries that contain algorithms or data structures connected to approximate pattern matching (Chapter 4).

To experimentally evaluate the different solutions we compiled a set of test instances (Chapter 5). We collected real word test data (natural language, DNA, and protein sequences) and analyzed them for several statistical properties. We furthermore designed and implemented a test instance generator that outputs sequences using as generating model a stochastic process (Bernoulli process, Markov process, or discrete autoregressive process), an approximate repeats model or Fibonacci sequences. An additional generator for search patterns allows providing complete artificial test instances for approximate pattern matching with controlled properties.

We finally performed a systematic experimental comparison of the different index structures and algorithms (Chapter 6). This is necessary and practically relevant because the theoretical running times are often not very meaningful in practice [NR02] (e. g., due to big constants hidden in the asymptotic notation). We first analyze the behavior of the individual solutions depending on the parameters of the input and then change the perspective to determine for several practically representative settings the respective best solution.
We end this thesis with a discussion and a description of open problems and possible extensions (Chapter 7). The implementations as well as the testing framework are available online for integrating and comparing other solutions in the future as well.
The appendix contains supplementary material such as a demo program, examples of real world and synthetic test instances, and detailed results of the experimental analysis (Appendix A).

1.6 Implementation environment

The implementation is realized in *SeqAn*[9], a popular and highly cited software library. The library contains data structures and algorithms for bioinformatics and is developed at the Freie Universität Berlin by Döring et al. [Dör+08] and Gogol-Döring and Reinert [GDR09], among others. The library is written in C++ and makes heavy use of template programming. Here we briefly give some information about the internals to better understand the design decisions for the implementation of the data structures and algorithms.
The overall goal is to produce efficient code by letting the compiler already do some of the work. This is achieved by using global functions with template parameters:

1. Type parameters: Most functions are implemented as general as possible so that they can be used with different input types. For each input type, a separate version of the function is generated at compile time which can be optimized for this specific type. (Type parameters begin with a capital letter `T`, e. g., `TString` for a string type.)

[8]"...implementing algorithms for use in a library is several times more difficult/expensive/time consuming/frustrating/ ...than implementations for experimental evaluation." [San09].
[9]Software library for **Seq**uence **An**alysis

2. Numeric parameters: Providing values for numeric parameters already at compile time enables the compiler to apply further optimizations.

3. Tag parameters: Switching between alternative variants of a function can be achieved by using so-called *tags*. The actual function call can this way already be resolved at compile time.

By defining these parameters already at compile time, the compiler can perform several optimizations much better, like inlining, loop unrolling, type-specific optimizations etc. The template mechanism is also used to emulate object-oriented inheritance by using so-called *template argument subclassing* [Dör+08] (which helps to avoid costly virtual function calls). Drawbacks of this generic approach are the longer compilation time, a more sophisticated programming style, lengthy declarations, and long error messages making debugging much more difficult [San09].

The most central class for our implementations is the string class for storing sequences: `String<TValue, TSpec>`. The parameter `TValue` defines the data type of the entries and can be among others:

- `char`: Character of one byte used to store regular text characters (e. g., of natural language text encoded in ASCII),
- `wchar_t`: Wide character spanning several bytes (on many systems it takes 4 B = 32 bit) to store strings of larger alphabets such as appearing, e. g., in Asian languages,
- `Dna`: Character storing one of the four bases of DNA sequences $\{ \mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T} \}$,
- `AminoAcid`: Character storing one amino acid to represent protein sequences,
- `bool`: for binary sequences,
- `unsigned int` etc.: for storing numeric tables, e. g., in index structures for strings.

The parameter `TSpec` defines how the string is stored, e. g., `Alloc<>` for storing it in main memory and `External<TConfig>` for storing it in external memory. The so-called *configuration* `TConfig` allows to define how many frames should be reserved for the string in main memory and how big the pages are (values that are crucial for the performance of external memory algorithms).

The library already contains several index structures and algorithms for strings and in particular for pattern matching (the contents are described in more detail in the dedicated Chapter 4, a short example program is in Section A.1).

Part II

Solutions

2 Index structures

This thesis focuses on offline approximate pattern matching, i. e., especially on index structures. This chapter describes several index structures for strings that form the basis of and can be used with algorithms for approximate pattern matching. The input of each index is either in a text t (of length n) or in a dictionary D (of cardinality l with a total concatenated length n). We present index structures using very different approaches including basic index structures (interesting for their concepts) as well as practical implementations and new developments (e. g., indexes optimized for secondary memory and compressed indexes). For each index we present the underlying idea, give some historic notes and argue why the index is interesting in our context; we furthermore describe several aspects that are outlined in the following paragraphs.

Data structure. The indexes are based on different kinds of data structures. Some indexes just consist of one or several simple tables of natural numbers, others are based on trees, and yet others use complex compressed data structures.
Most indexes consist of several components (also called *fibres* in the implementation), which in some cases can be constructed or used independently of each other. (Components that are not relevant for pattern matching but can be used for other purposes are not described here.)
Some indexes depend on parameters that influence the resulting data structure. These parameters can, for example, choose between several possibilities for underlying data structures, set the external memory configuration, or specify a compression ratio. The parameters of the data structure are summarized in boxes labeled "index parameters".
The space consumption of an index structure X is denoted by S_X. The space consumptions of the described indexes vary between the size of the text (or even the size of the compressed text) and many times the text size (e. g., $20\,n\,\mathrm{B}$)[1].
Some indexes are limited to a maximal text length, e. g., because the layout of the data structure only permits a fixed number of bits for some values. In these cases we explicitly indicate the maximal text length. In practice however, the maximal text length might for some indexes additionally be limited by other factors, such as the space consumption when used with a limited main memory. We indicate this limitation in these cases.

Construction algorithm. Each index comes with one or several variants of construction algorithms that build the data structure before it can be used to answer pattern matching queries. In some cases, the components of the index can be constructed independently of each other. This makes it possible to reuse some algorithms for other indexes that contain the same or similar components.
The construction algorithms depend in some cases on additional parameters that only affect the construction and not the resulting data structure. These parameters can be, for example, buffer sizes, sorting strategies, etc. and are summarized in boxes labeled "construction parameters".
The worst case running time of a construction algorithm for an index X is denoted by $T_X^{\text{construct}}$. The construction algorithms have worst case running times between $\mathcal{O}(n)$ and $\mathcal{O}\left(n^2 \log n\right)$. However, algorithms with $\mathcal{O}(n)$ might in practice be slower than, for example, an $\mathcal{O}\left(n^2\right)$ algorithm that takes secondary memory into account and is optimized to that effect.
Some algorithms need more space during the construction which can be freed once the index has been built completely. The maximum total space used during construction is called *construction memory* and denoted by $S_X^{\text{construct}}$. It can in some cases be several times the space of the index itself. The value is given only if it differs from the space of the completely built index.

[1] The symbol B denotes a byte = 8 bit.

Search. All index structures permit to perform exact pattern matching and we describe the algorithms to answer the different types of queries for a given search pattern p of length m (see Section 1.4): boolean, counting, and position query. This corresponds to evaluating the functions $R_{\text{bool}} : \Sigma^* \to \mathbb{B}$, $R_{\text{count}} : \Sigma^* \to \mathbb{N}$, and $R_{\text{pos}} : \Sigma^* \to \mathcal{P}(\mathbb{N})$ after the index structure has been completely built. For some data structures it is faster to answer only a boolean or counting query as compared to a position query. The worst case running time to answer the three types of queries using an index X are denoted by T_X^{bool}, T_X^{count}, and T_X^{pos}. Since the number of matching occurrences is frequently contained in the analysis, it is abbreviated as $occ := \left|R_{\text{pos}}\right|$. The worst case running time for the queries is in some cases $\mathcal{O}(m)$ (optimal), while some indexes need $\mathcal{O}(n)$ but might still be very fast in practice.
A lower bound to answer exact pattern matching queries in a text for a pattern of length m is $\Omega(m)$ since all characters of the pattern have to be examined. If all positions have to be output, the lower bound is therefore $\Omega(m + occ)$.
The additional space usage for performing exact pattern matching is negligible for all indexes, never more than $\mathcal{O}(m)$, and therefore not mentioned individually.
Some algorithms are limited to a minimal or maximal pattern length m; this is indicated for the respective search algorithms.

(Algorithms for *approximate* pattern matching are not described and analyzed in this chapter, but separately and in more detail in Chapter 3.)

Analysis. For indicating the space and time consumption of the indexes we have to decide between many alternatives. Is it more useful to give worst case or expected case values? How much space is assumed to be needed for storing a text character? Is the size of a computer word constant or variable? Should the space consumption be indicated in byte or in bit?

We decided to give the worst case value for space consumption and running times and where available additionally also an indication of the expected value. Some solutions are not fully specified and allow for different implementations, so it is not possible to give exact values for the space consumption or running times. In these cases we attempt to give asymptotic bounds.
We assume that the alphabet size is constant (if not stated otherwise), which is reasonable since many important applications deal with biological data where usually $\sigma \leq 30$. For the space consumption we furthermore assume that a text character can be stored in one byte. (This does not hold for applications with larger alphabet such as, e. g., for Chinese texts where a character needs several bytes, or for applications where several characters are stored within one byte.)
The space consumption is given either as exact value (omitting constant terms) and otherwise asymptotically. Even if the text length is in some cases limited to a maximum value (see above), we investigate the asymptotic behavior since it can best illustrate the growth of the functions.
We assume the machine word size $W \in \mathbb{N}$ is constant, we are working on a 32-bit machine with $W = 32\,\text{bit} = 4\,\text{B}$, and that an integer value is stored in 32 bit as well. To permit storing a reference to a text position in one machine word, we assume $n \leq 2^W \Leftrightarrow W \geq \log n$.
The space consumption is given in B (byte) for all but the compressed indexes, where it is given in bit because it is more expressive when handling compressed data [NM07].

Implementation. The software library SeqAn already contained implementations of several of the described index structures, this is indicated in each section. We implemented some more index structures that offer different trade-offs of the space consumption, construction time, query time, and/or secondary memory usage.
An index in SeqAn is derived from the class `Index<TText, TIndexSpec>`, where `TText` defines the type of the underlying string and `TIndexSpec` is used to specialize the subclass.

The components (fibres) of an index object `index` can be constructed using the function `indexCreate(index, FibreTag)`, where `FibreTag` defines which component to create. This is, however, not necessarily required explicitly because there is an automatic mechanism that creates a component as soon as it is used for the first time.
Some of the indexes can be saved to disk by using the function `save(index, filename)`. It can then be loaded to memory again with the function `open(index, filename)`.

Searching in a text or dictionary can be achieved by using two classes: the `Finder` and the `Pattern` class. An object of type `Finder<THaystack, TFinderSpec>` stores information about the text or the index structure (which are summarized by the term *haystack* and the template type `THaystack`). An object of type `Pattern<TNeedle, TPatternSpec>` stores information about the search pattern (which is also called *needle* and of template type `TNeedle`). The type parameters `TFinderSpec` and `TPatternSpec` define specializations of the finder and pattern class to be able to use different search algorithms. To perform exact pattern matching the function `bool find(finder, pattern)` can be used and called repeatedly as long as it returns true. The matching positions of the occurrences in the text can be retrieved with the function `position(finder)` (the matches are thereby allowed to be returned in arbitrary order). To execute a boolean query, the `find` function can simply be called only once. To execute a counting query there is no direct efficient way in the implementation that works with all index structures, apart from calling the `find` function repeatedly (a short example program is in Section A.1).
The suffix tree index structures (see below) additionally provide a tree iteration interface by implementing the class `Iterator<TIndex, TopDown<> >` with functions to navigate through the tree nodes (e. g., `goDown`, `goRight`, `goUp`, `goRoot`, etc.).

Discussion. Each index structure has its advantages or disadvantages depending on the space and time consumption or secondary memory usage, and behave differently depending on the alphabet size, text length or type of search query, among others. Some index structures also provide additional functionality, such as an efficient traversal of suffixes, or substrings of the text. These aspects are discussed for each index structure.

2.1 Suffix arrays

The *suffix array* is a simple, yet very important data structure for pattern matching, and also for string processing in general. It was developed by Manber and Myers [MM90; MM93] and has also been introduced under the name *PAT array* by Gonnet et al. [Gon+92] (it is abbreviated as *SA* and sometimes as *Pos* [Gus97]).
The suffix array can be used as an index structure *for a text*. To use it as index *for a dictionary*, the entries of the dictionary can be concatenated to form one text (as described in Section 1.3).
The suffix array is interesting in our context for several reasons. The data structure itself is very simple, being just one array of integers. Furthermore, it is relatively small compared to the other index structures that are discussed in the following sections. It forms the basis for many other data structures: some make use of the underlying concepts, while others directly include the suffix array as one component.
There are several variants of suffix arrays, some of which are described in the following sections, starting here with the very basic suffix array data structure itself.

2.1.1 Classical suffix array

The concept of a suffix array is as follows: The suffixes of the given text are sorted lexicographically, and the starting positions of the suffixes are stored in an array in increasing lexicographic order.

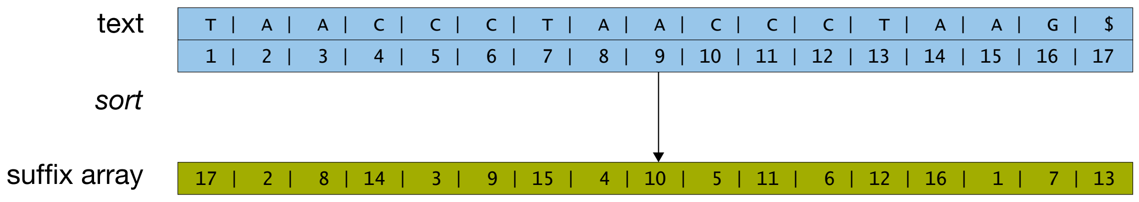


Figure 1: Suffix array: data structure with schematic example.

The positions of the text t = "TAACCCTAACCCTAAG" are sorted according to the lexicographic ordering of the text suffixes.

This sorted array already makes it possible to solve several pattern matching problems quite efficiently as discussed below.

2.1.1.1 Data structure

Formally the suffix array *SA* stores a permutation of the set $\{1, \ldots, n\}$ of all starting positions of the text. The suffixes are sorted according to the lexicographic ordering $\prec$ (see Section 1.2), i. e., for each entry $1 < i \leq n$ it holds that $i - 1 \prec i$ which is by definition equivalent to $t_{[SA[i-1]\,..]} \prec t_{[SA[i]\,..]}$. An implementation of the suffix array was already available in the software library. (To use the suffix array for pattern matching, it is available as part of the *enhanced suffix array* `IndexEsa`. More details regarding the implementation and parameters are therefore given in the corresponding Section 2.2.6.)

The suffix array requires linear space $S_{\mathrm{SA}} = \mathcal{O}(n)$, and also the constants are small: If a text character is stored in 1 B and a starting position is stored in a word of 4 B = 32 bit, the suffix array needs $S_{\mathrm{SA}} = 4\,n\,\mathrm{B}$.
The text length is only limited by the size of a word to store the text positions. The maximum text length that can be indexed with a suffix array using 32 bit words is $2^{32} = 4\,294\,967\,296 \mathrel{\widehat{=}} 4\,\mathrm{GiB}$.

Index parameters of `IndexEsa` (all variants):

1. `TIndexStringSpec`: This parameter chooses an implementation for the storage of the index tables, e. g., whether the tables should be held in main memory or in secondary memory. Possible choices are the string types, e. g., `Alloc`, `External<TConfig>`, or `MMap<TConfig>`, see Section 1.6.
(This parameter can not be passed directly, but has to be set by overloading a meta-function, e. g., `DefaultIndexStringSpec`.)

2.1.1.2 Construction

The task "*Given a text t, construct the suffix array of t.*" might seem very simple. However, many dozens of algorithms to solve this task have been devised during the last twenty years. Several algorithms to construct suffix arrays are included in the software library:

- `SAQSort`: This simple approach uses an existing comparison based sorting algorithm (quick sort in the implementation), together a with a user-defined comparison function for a lexicographic comparison of the starting positions. This approach takes $\mathcal{O}(n^2 \log n)$ time in the worst case, since each comparison can take $\mathcal{O}(n)$ steps.

Algorithm tag	Reference	$T_{\mathrm{SA}}^{\mathrm{construct}}$	$S_{\mathrm{SA}}^{\mathrm{construct}}$
`SAQSort`	Quicksort	$\mathcal{O}\left(n^2 \log n\right)$	$\mathcal{O}(n)$
`ManberMyers`	Manber and Myers [MM93]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
`LarssonSadakane`	Larsson and Sadakane [LS07]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
`BwtWalk`	Baron and Bresler [BB05]	$\mathcal{O}\left(n\sqrt{\log n}\right)$	$\mathcal{O}(n)$
`Skew3`	Kärkkäinen and Sanders [KS03]	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`Skew7`	Weese [Wee06]	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 1: Suffix array: Construction algorithms with their worst case running times.

(Based on [Pug+07].)

- `ManberMyers`: Together with the data structure, Manber and Myers [MM93] in 1993 proposed a construction algorithm running in $\mathcal{O}(n \log n)$ worst case time. This algorithm is also called *prefix doubling algorithm* but is of little practical importance now [Pug+07].
- `LarssonSadakane`: Larsson and Sadakane [LS07] developed *qsufsort* in 2007, another *prefix doubling algorithm* having the same worst case running time, but outperforming the algorithm by Manber and Myers [MM93] in practice [Pug+07].
- `BwtWalk`: Baron and Bresler [BB05] propose using the Burrows-Wheeler transform for sorting the starting positions of suffixes. (The implementation of the algorithm is described by Gogol-Döring and Reinert [GDR09].)
- `Skew3`: The *skew algorithm* by Kärkkäinen and Sanders [KS03] from 2003 is a direct linear time construction algorithm for suffix arrays. It uses a so-called *difference cover (DC)* of 3 [Kär+06].
- `Skew7`: A modification by Weese [Wee06] of the skew algorithm uses a difference cover of 7, aimed at a better practical performance. This is the default suffix array construction algorithm.

The theoretical worst case times are not very meaningful in practice since several super-linear algorithms perform better than other linear-time algorithms on real-world texts. The survey by Puglisi et al. [Pug+07] gives an overview of many important algorithms, compares the running times and space usages, and gives experimental results. For practical applications, trade-offs can be made regarding the average running time, the worst case running time and the working memory needed. The most efficient construction algorithms of the implementation are also described in more detail by Weese [Wee12].
A more recent survey by Dhaliwal et al. [Dha+12] focuses on newer trends for construction algorithms, especially using less working memory or working in external memory or with compression techniques.
A guide for suffix array construction directed at bioinformaticians has recently been published by Shrestha et al. [Shr+14].

(The software library *libdivsufsort* by Mori [Mor08] provides an interface to several of the most important suffix sorting algorithms. It furthermore contains experimental results for several popular text corpora.)

Construction parameters of `IndexEsa` (suffix array):

1. `TAlgSpec`: Algorithm to create the suffix array (see Table 1). (The default algorithm is `Skew7`, the modified version of Kärkkäinen and Sanders [KS03].)
2. (Algorithm specific parameters if applicable.)

2.1.1.3 Search

The suffix array can speed up the search of a pattern in a text. We start the explanation with two simple observations:

1. We are searching for substrings (= infixes) of the text, and each infix is a prefix of a suffix of the text.
2. The suffix array enables us to search for prefixes of suffixes, since it stores their starting positions in lexicographic order.

The starting positions of occurrences of a pattern p are stored in a consecutive interval (possibly empty) in the suffix array, we just have to determine its borders [Gus97, Section 7.14]. To find the left border we perform a simple binary search in the suffix array. For each comparison we compare the pattern with the characters of the text at the given starting position (in case of equality we decide to continue in the left subinterval of the recursion). To find the right border of the interval we can do the same, but in case of equality continue in the right subinterval.
The binary search takes $\mathcal{O}(\log n)$ steps and each step takes $\mathcal{O}(m)$ time in the worst case, yielding a search time of $T_{\mathrm{SA}}^{\mathrm{bool/count}} = \mathcal{O}(m \log n)$ and $T_{\mathrm{SA}}^{\mathrm{pos}} = \mathcal{O}(m \log n + occ)$ in the worst case.

A simple heuristic can be used to reduce the number of redundant character comparisons. It is called *mlr-heuristic* [Gus97] and maintains two additional values during the binary search: lcp_{l} and lcp_{r}. They are in each step set to the length of the longest common prefix (abbreviated as *lcp*) of the search pattern and the left interval border $lcp_{\mathrm{l}} := \mathrm{lcp}(p, t_{[SA[l]\,..]})$, and of the right interval border $lcp_{\mathrm{r}} := \mathrm{lcp}(p, t_{[SA[r]\,..]})$, respectively. In the next step of the recursion the algorithm does not have to compare the pattern from the beginning, but the comparison can skip the first $\min\{\, lcp_{\mathrm{l}}, lcp_{\mathrm{r}} \,\}$ characters. This does not improve the asymptotic worst case running time, but it reported to perform well in practice [Gus97]. In the implementation, the mlr-heuristic is the default search algorithm for a suffix array.

2.1.1.4 Discussion

The suffix array is a very simple data structure (just one table) that also allows for an efficient construction, in theory, as well as in practice on a real computer and for large inputs. Due to its simplicity it is, however, limited in functionality (compared to other indexes such as the suffix tree, Section 2.2.2) and in the asymptotic search efficiency.
The suffix array can be extended with additional extra tables to enable a more efficient search and extended functionality. One such table is the LCP table that stores information about longest common prefixes. The suffix array can even be extended with two other tables to replace a suffix tree while keeping all asymptotic times; this is described in Section 2.2.6 after the suffix tree has been presented. Yet another related table is the *inverse suffix array* SA^{-1}, defined as follows: $SA^{-1}[i] = j$ if $SA[j] = i$, which can be computed in linear time.
The suffix array forms the basis or has a close connection to many other index structures for strings: It corresponds to the sorted set of leaves of a suffix tree (Section 2.2.2), is very similar to the *positions* table of a q-gram index (Section 2.4.1), has a close connection to the

Burrows-Wheeler transform (Section 2.2.6), and forms the basis of many compressed indexes (Section 2.3).

An extension of the basic suffix array and a parallel algorithm for approximate pattern matching in external memory was proposed by Cheng et al. [Che+03]. A suffix array for only the words of a (e. g., natural language) text was proposed by Ferragina and Fischer [FF07].

2.1.2 Suffix array with LCP table

The suffix array can be extended with stored information about the longest common prefixes to achieve a better asymptotic worst case running time [MM93; Abo+04].

2.1.2.1 Data structure

The additional table *lcptab* of length n stores the length of the longest common prefix of two consecutive entries in the suffix array [Abo+04] (it is also called *Hgt* [MM93] or *Height array* [Kas+01]). The LCP table is formally defined as $lcptab[0] := 0$ and $lcptab[i] := \mathrm{lcp}(SA[i-1], SA[i])$ for $1 \leq i \leq n$. It requires $4\,n\,\mathrm{B}$ on a 32-bit machine. The resulting space consumption of the index is $S_{\mathrm{SA\,+\,lcp}} = 8\,n\,\mathrm{B}$.
In the implementation, the LCP table is part of the enhanced suffix array described in Section 2.2.6.

2.1.2.2 Construction

An efficient algorithm to construct the LCP table in linear time using the already constructed suffix array and based on the work of Manber and Myers [MM93] is proposed by Kasai et al. [Kas+01] with $T^{\mathrm{create}}_{\mathrm{SA\,+\,lcp}} = \mathcal{O}(n)$. This algorithm (called `KasaiOriginal`) and a pipelined version (called `Kasai`) are contained in the software library. Later this algorithm has been improved by Manzini [Man04] to save working memory. Recent algorithms to construct the LCP table are by Gog and Ohlebusch [GO11], Fischer [Fis11], and Bingmann et al. [Bin+13].
The approach by Manber and Myers [MM93] finally transforms the LCP table in linear time, so that the *information needed in the binary search* (and not only for consecutive entries) is available in constant time [Fis+06].

Construction parameters of `IndexEsa` (suffix array with LCP table):

1. `TAlgSpec`: Algorithm to create the LCP table: `KasaiOriginal` or `Kasai` (default).

2.1.2.3 Search

With the LCP information, the asymptotic worst case time for exact pattern matching can be reduced to $T^{\mathrm{bool/count}}_{\mathrm{SA\,+\,lcp}} = \mathcal{O}(m + \log n)$. However, in practice, the mlr-heuristic has been shown to have comparable performance and only needs constant additional memory during the search [MM93; Gus97; Wee12].

2.1.2.4 Discussion

The suffix array together with the LCP table already allows for simulating a bottom-up traversal of the corresponding suffix tree [Kas+01; Abo+04] and to efficiently solve other problems such as frequency-related data mining in databases of strings in linear time [Fis+06].
The suffix array and the LCP table are also used as part of the enhanced suffix array (with the additional *child table*) to simulate all functionality of a suffix tree with asymptotically the same time bounds, so that searching takes only optimal $\mathcal{O}(m)$ time in the worst case (Section 2.2.6).

2.2 Tries and suffix trees

2.2.1 Trie

The *trie*[2] (also known as *digital search tree*) is one of the basic and conceptually in our view most important data structures for pattern matching. A trie is also sometimes referred to as *prefix tree* since it is a *tree* data structure and stores common *prefixes* of strings together.
With a trie it is possible to store and/or to index a dictionary (i. e., a set of strings). If the problem instance is one long text, it is possible to index all suffixes of the text in the trie (which is then called *suffix tree* and is treated separately in the following Section 2.2.2). In the case of a text which is structured as words, it is also possible to index all words of the text in the trie; searches can then be carried out only for prefixes of words and not for arbitrary substrings.

2.2.1.1 Data structure

The trie of a set of strings D is a rooted tree where each edge is labeled with one character of the alphabet Σ. All outgoing edges of a node are required to have different labels. Each node represents the string which is formed by concatenating the characters on the path from the root node (this string is called the *path label* of the node; the node with path label r is be denoted by $\overline{r}$). All descendants of a node with path label r therefore have path labels sharing the common prefix r. A trie of a dictionary D contains nodes for those and only those nodes that represent a prefix of a (or a complete) string $s_i \in D$, the root node represents the empty string.
A node with a path label s_i that is an element of the underlying dictionary ($s_i \in D$) is called *terminal*. To simplify the explanation and to obtain a one-to-one correspondence between leaves and terminal nodes, each string of D is appended with a special terminal character $\$ \notin \Sigma$ that is not contained in the alphabet. This ensures that no string in the dictionary is prefix of another string of the dictionary and the edges are then labeled with characters from $\Sigma_\$:= \Sigma \cup \{\,\$\,\}$. This modification does not change the asymptotic space usage and running times. The term *leaf* rather than *terminal node* is used in the following exposition. An example of the trie data structure is shown in Figure 2.
A trie needs $S_{\text{Trie}} = \mathcal{O}(n)$ space in the worst case because each character of the strings in the dictionary needs at most one additional node of constant size and its incoming edge.

When implementing a trie there are several possible choices. Some choices are more relevant when the trie is used as a suffix tree; however, approaches that can be used with any trie are already presented here. A trade-off between space and running time can be made depending on the storage of the children of a node (let t_{step} denote the time to follow an edge from a node to a child with a given character):

- Sorted list: Using binary search a step needs $t_{\text{step}} = \mathcal{O}(\log \sigma)$ worst case time.
- Unsorted list: Using linear search a step needs $t_{\text{step}} = \mathcal{O}(\sigma)$ worst case time.
- Array of size σ where each position of the array corresponds to one character of the alphabet: A step then needs $t_{\text{step}} = \mathcal{O}(1)$ worst case time, the space usage therefore grows by the factor σ in the worst case.
- Hash table of size $\mathcal{O}(|\mathit{children}|)$ and a perfect hash function [GV05]: A step then needs $t_{\text{step}} = \mathcal{O}(1)$ worst case time, and during the construction the additional time of $\mathcal{O}(|\mathit{children}| \log |\mathit{children}|)$ is needed to compute the hash function.

[2]The term *trie* originally comes from the field of information re*trie*val [Fre60].

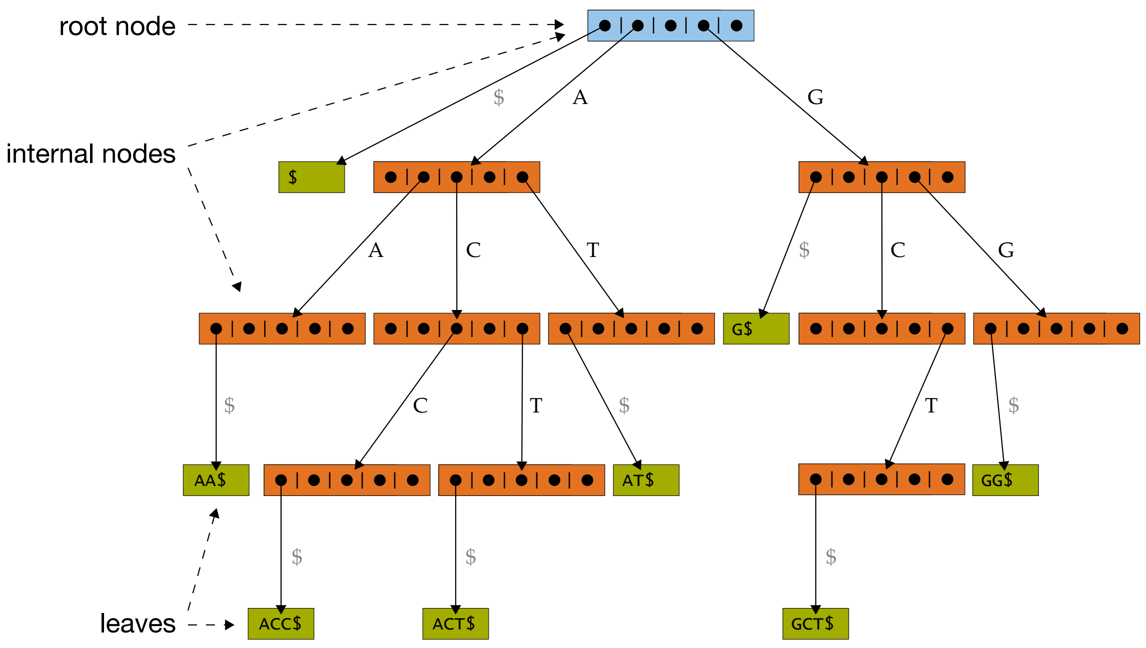


Figure 2: Trie: data structure.

A trie for the dictionary $D = \{\epsilon, \text{“AA”}, \text{“ACC”}, \text{“ACT”}, \text{“AT”}, \text{“GCT”}, \text{“GG”}, \text{“G”}\}$. Each entry $s_i \in D$ is terminated with a special $ symbol to achieve a one-to-one correspondence between leaves and terminal nodes. In this example, the nodes store child pointers using an array of fixes size (with entries corresponding to the characters A,C,G,T, and $).

(The asymptotic terms are given as a function of the alphabet size σ here to illustrate the differences of the approaches. In the rest of this work the alphabet size is usually considered to be constant if not stated otherwise.)

There are several similar variants of compacted tries (named *compact prefix tree*, *radix tree*, or *Patricia trie*[3] by Morrison [Mor68]). In these variants, the number of internal nodes is reduced by contracting paths of nodes with only one child (in the example of Figure 2 the nodes $\overline{\text{“GC”}}$ and $\overline{\text{“GCT”}}$ would be combined into one node). The edges are then labeled with *strings* instead of single characters. In most cases the labels are not stored explicitly as strings; instead a start position of the label in the input is stored, together with the end position or the length of the label. This reduces the asymptotic worst case space usage: Each edge only needs constant space in this setting, and by construction each internal node has at least two children. Since there are in total $l := |D|$ leaves, there cannot be more than l internal nodes, resulting in a total worst case space usage of $\mathcal{O}(l)$ (compared to $\mathcal{O}(n)$ as in the not compacted version).

There are many more possible choices when implementing tries. The proposed approaches differ, for example, in which information is actually stored inside each internal node; the not explicitly stored information is computed later on the fly. The information stored in a node can include:

- the length of the label of the incoming edge,
- a starting position of the label within the input,
- the corresponding end position within the input,
- the depth within the tree (counted in characters),
- the first character on the incoming edge,
- a link to the first child,
- a link the right sibling,
- a link to the first leaf in the subtree of the node, etc.

[3]Patricia = **P**ractical **A**lgorithm **t**o **R**etrieve **I**nformation **C**oded **i**n **A**lphanumeric

Depending on the application, also the *leaves* can store additional information, such as a pointer to the corresponding string of D (or the starting position if the problem instance is a text). Furthermore, the different proposals try to squeeze the stored information in the internal nodes and in the leaves in as few bits as possible. For the index structures based on tries (especially for the variants of suffix trees), we point out in the following sections which information is actually stored inside the internal nodes and how.

Another common approach is to use a binary representation of the input strings instead of the input strings themselves. This results in an alphabet of size 2, which makes is possible to simplify the data structure because each node has at most two children (or even *exactly* two children when using a compacted variant).

2.2.1.2 Construction

A trie without contracted edges can be built for a given dictionary D by inserting the strings $s_i \in D$ one after the other into the tree structure. The algorithm therefore searches s_i in the trie (see the following subsection) and if an edge is missing, a new node with corresponding incoming edge is created. Building a trie for a dictionary of concatenated length n therefore takes in total $T_{\text{Trie}}^{\text{construct}} = \mathcal{O}(n)$ worst case time.
To create a Patricia trie the above algorithm has to be modified slightly. When inserting a string and a corresponding outgoing edge is missing at some point, the algorithm might have to split an existing edge and create a new internal node. However, this does not change the overall asymptotic running time.

2.2.1.3 Search

To perform exact pattern matching with a trie (i. e., to find out whether a given search pattern p occurs in the dictionary), the algorithm processes p character by character. It starts at the root node, identifies in each step the correct outgoing edge and descends to the corresponding child. If the pattern has been fully processed and the current node has an outgoing edge labeled with \$ to a leaf, the algorithm returns true. If in one step during the descent no corresponding outgoing edge exists, the pattern does not occur in the dictionary and the algorithm returns false.
Performing exact pattern matching of a pattern with length m in a dictionary using a trie needs $T_{\text{Trie}}^{\text{bool/count/pos}} = \mathcal{O}(m)$ optimal time in the worst case, independent of the size of the underlying dictionary.
It is also possible to perform a prefix search (determine whether or where the search pattern p occurs as a prefix of a string $s_i \in D$). Therefore the search algorithm is slightly modified: If the descent was successful and the search pattern has been fully processed, it simply outputs all leaves of the current subtree (or the information stored inside the leaves). This takes additional time proportional to the number of matches, yielding a worst case search time of $\mathcal{O}(m + occ)$.
When searching a Patricia trie, the search algorithms have to be slightly modified because at the end of the descent, the algorithm might not stop at a node, but in the middle of an edge. If we are looking for a complete match in the dictionary, the algorithm returns false. If we are performing a prefix search, the algorithm outputs the occurrences in the subtree rooted at the node on the lower end of the current edge.

2.2.1.4 Discussion

One drawback of tries is the linked structure, which can consume much memory in theory as well as in practice: One single character in the input (stored in 1 B) can require the creation of a new node, which contains two or more pointers (each of size 4 B on a 32-bit machine) and possibly more data.

Skip/count or blind search. Another variant of tries stores for each edge the first character of the label and its length instead of the starting position (in this variant, the leaves therefore need to store the starting positions). To perform a search in this trie, the algorithm again descends from the root node. It then retrieves the outgoing edge corresponding to the first character of the pattern and follows that edge blindly to the child node.[4] In the search pattern the algorithm simply skips the remaining characters of the edge label (whose length is stored in the node). This is repeated until the algorithm reaches the end of the pattern.

Since the algorithm followed some characters of the edge labels blindly, it finally has to check whether or not the found node is an actual match. This is done by extracting the starting position of one of the leaves of the subtree and by comparing the text at that position with the search pattern. This verification has to be done only once for all leaves because the leaves of the subtree are either all matches or all no matches, since they share the common prefix.

This variant needs asymptotically the same worst case time to perform a search. In practical implementations it can, however, be faster because the algorithm does not need to access the underlying dictionary or text during the descent, where each comparison might need a random memory access or even an I/O (input/output) operation if the input is stored in external memory. With the skip/count trick the text is only accessed once. This is used, for example, in the DiGeST index by Barsky et al. [Bar+08] and called *blind search* (Section 2.2.7). Additionally, this variant can save some space in practical applications if a character of the alphabet is stored in less space than a pointer.

Variants of tries. There have been some proposals on how to make tries perform better in practice. *Ternary Search Tries* by Bentley and Sedgewick [BS98] are a practical compromise between the time efficiency of tries and the space efficiency of binary trees [Kru08]. *String B-trees* by Ferragina and Grossi [FG99] are a combination of B-tree and tries, that gives good theoretical bounds for I/O operations; however, we know of no efficient construction algorithm in practice [Bar+08]. *Burst tries* by Heinz et al. [Hei+02] reduce the space consumption of tries by replacing subtrees with other more compact containers. *Cache-Oblivious String Dictionaries* by Brodal and Fagerberg [BF06] are a theoretical proposal for making tries cache-oblivious, and *HAT-tries* by Askitis and Sinha [AS07] are a practical approach making burst tries cache-oblivious. *cedar* by Yoshinaga [Yos12] is an efficient implementation of a trie and comes with an experimental comparison of several other state-of-the-art trie implementations.

In SeqAn there is an implementation of a class `Trie` as a subclass of the class `Graph`. However, it is not a subclass of `Index` and not primarily intended for pattern matching. The index structures described in the following sections are better suited for this task.

2.2.2 Suffix tree

The *suffix tree* is one of the most popular data structures for strings. It allows for (sometimes surprisingly) efficient solutions for a variety of text problems [Gus97]. The suffix tree for a given text t is the Patricia trie for all suffixes of $t\,\$$ (the text is terminated by a special $\$ \notin \Sigma$ symbol for technical reasons). Each leaf stores the starting position of its corresponding suffix and each edge conceptually stores the start and end positions of an occurrence of its label in the text.

[4] This method is essentially the skip/count trick used in Ukkonen's algorithm to build a suffix tree [Gus97, Section 6.1.3].

It is also possible to use a suffix tree as index for a *dictionary*, i. e., for a set of strings instead for only one string. To do so, every string $s_i \in D$ is appended with a distinct new terminal symbol $\$_i \notin \Sigma, i \in [1, l]$ and these strings are concatenated to yield one string over the alphabet $\Sigma \cup \{ \$_1, \ldots, \$_l \}$ (as described in Section 1.3.4). The suffix tree of this string is called *generalized suffix tree* for the dictionary D and permits to also find substrings of the stored strings (and not only prefixes).
A short history of the suffix tree data structure, its classical construction algorithms, and several applications can be found in the book of Gusfield [Gus97].

2.2.2.1 Data structure

The suffix tree for a string of length n occupies $S_{\text{Suffix tree}} = \mathcal{O}(n)$ space in the worst case because the trie has n leaves, each internal node has at least two children, and the nodes and edges both need only constant space.
In the implementation of suffix trees we have basically the same choices as for tries (how to store the references to the children, contents of the nodes and leaves etc., see Section 2.2.1). Additionally, some proposed suffix trees influence the order in which the nodes are stored: If the nodes are stored in a clever ordering, it is possible to exploit some regularities within the suffix tree and to save some space. This is used in some of the approaches in the following sections and mentioned there.
The concept of *suffix links* is used in some algorithms, especially for the construction of the tree. These algorithms store an additional pointer (the suffix link) inside each internal node: it points from each node $\overline{ur}$ to the node $\overline{r}$ where the first character is removed from the path label ($u \in \Sigma_\$, r \in \Sigma_\$^*$).

A few observations regarding the nodes of the suffix tree are [Gus97]: The leaves of the suffix tree correspond to the suffixes of the text. If the children of each node are sorted lexicographically from left to right, the leaves of the suffix tree are essentially the entries of the suffix array. The internal nodes of a suffix tree correspond to substrings of the text that occur more than once. Each internal node of the suffix tree corresponds to an interval of the suffix array (but not the other way around).

2.2.2.2 Construction

To construct a suffix tree of a text t, a naive algorithm inserts the suffixes one after the other into an initially empty trie. This needs quadratic time $\mathcal{O}(n^2)$ in the worst case, since inserting one suffix takes $\mathcal{O}(n)$ time and there are n suffixes.
However, in the special case when building a trie for all suffixes of a text, there are more efficient algorithms. The first algorithm to build suffix trees in linear time $T^{\text{construct}}_{\text{Suffix tree}} = \mathcal{O}(n)$ was given by Weiner [Wei73] in 1973 and Donald Knuth is cited to have called it "*Algorithm of the Year 1973*" [Gus97]. A few years later McCreight [McC76] proposed a more space-efficient algorithm also running in linear time. In 1995 Ukkonen [Ukk95] presented his construction algorithm, which is simpler to describe and among the most popular algorithms for suffix trees. In 1997 Farach [Far97] presented an algorithm to build suffix trees for texts with a large alphabet (whereas in the other algorithms the alphabet is usually considered to be of small constant size).
In the last decade, many algorithms have been developed that take into account more practical aspects, like the behavior in main or external memory. Some of these algorithms are described in the following sections.

2.2.2.3 Search

With a suffix tree it is possible to efficiently locate all occurrences of a pattern in a text. Since a suffix tree is essentially a trie of the suffixes, we can perform a prefix search in that trie to find arbitrary substrings of the text. The search algorithm works basically the same way as for a trie. Starting at the root node, the algorithm processes the pattern from left to right and follows the edges corresponding to the characters. Since the labels are not stored explicitly, but we only have the start and end position of the label, we have to carry out the comparisons in the underlying text.
When the search ends at an internal node (in case it ends in the middle of an edge, we take the node at the lower end of the edge), the algorithm iterates over the subtree below that node. For each leaf in this subtree, the starting position of the corresponding suffix is returned. If during the descent an edge corresponding to the characters of the pattern was missing, the text does not contain the pattern and the algorithm returns the empty set.
Performing exact pattern matching of a pattern with length m in a text (or in a dictionary) using a suffix tree needs $T^{\text{bool}}_{\text{Suffix tree}} = \mathcal{O}(m)$ and $T^{\text{count/pos}}_{\text{Suffix tree}} = \mathcal{O}(m + occ)$ time where occ is the number of results. This is optimal and independent of the size of the text.

2.2.2.4 Discussion

The suffix tree is a very important data structure for pattern matching, in particular because it takes only linear space and can also be built in linear time. In addition, it is a very versatile data structure that offers additionally functionality compared to, e. g., the suffix array. It is possible to perform a traversal of the tree nodes (corresponding to substrings of the text), either in top-down or in bottom-up order. It can be used in numerous applications, and "*Gusfield devotes about 70 pages of his book to applications of suffix trees*" [Abo+04; Gus97].
There are many variants for possible implementations of suffix trees that additionally come with specialized construction algorithms. These ideas are described in the following sections. The ideas to improve tries presented above can furthermore often be transferred to suffix trees as well.
However, the constants hidden in the asymptotic notation of the linear space consumption are quite big for some practical applications. The suffix tree of Kurtz is considered to be the most compact classical representation, but still uses on average 10 B for each character of the input (20 B in the worst case, see Section 2.2.3). This means that a text of 800 MiB already occupies 8 GiB of memory, making it impossible to be stored in the main memory of a regular desktop computer.
Furthermore, a direct implementation of the tree structure does not work very well together with secondary memory and caches because many random memory accesses occur during the traversal due to the poor locality of reference [Abo+04]. If a suffix tree does not fit into main memory, the performance therefore degrades drastically. There are suffix tree representations optimized for the use in external memory that apply a partitioning of the tree into a so-called *suffix forest* of smaller trees (Section 2.2.7).

Suffix links are primarily used in linear-time construction algorithms for suffix trees [Gus97], while several modern implementations use other construction algorithms (see below) and do not contain suffix links to reduce the space consumption. Even though suffix links can be used to solve some problems more efficiently [Abo+04], they are not necessary in many applications, such as pattern matching.

2.2.3 Space reduced suffix tree by Kurtz

The suffix tree proposed by Kurtz [Kur99] is a classical implementation of the data structure. It is known for a very compact representation of the tree nodes, based on intelligent ‘‘bit squeezing’’ and more optimizations based on regularities in the tree.

2.2.3.1 Data structure

The nodes of the suffix tree are stored in two tables, one for the internal nodes, and one for the leaves. The internal tree nodes store the following information:

- pointer to the first child (29 bit),
- pointer to the next sibling (29 bit),
- a bit marking a node as rightmost child,
- suffix link (29 bit, stored in the next sibling field of the rightmost child which would otherwise be empty),
- the head position (27 bit): the starting position of the suffix that caused the creation of the node (can together with the depth be used to retrieve the incoming edge label),
- the string depth in the tree (27 bit).

The nodes are thereby stored in clever ordering, such that the head position of many consecutive nodes (called *node chain*) can be calculated from the head position of the previous nodes. This makes it possible to distinguish two types of nodes: small nodes (consisting of two 32 bit integers) and large nodes (consisting of four 32 bit integers). Additionally, only the value *dist* needs to be stored which indicates the distance to the first node of the node chain. This ordering of the nodes makes it possible to save half the space of a node in many cases.
Another additional optimization distinguishes large nodes in *high nodes* and *regular nodes*, based on their depth in the tree: Nodes with depth $\leq 2^{10}$ use only 10 bit for storing the depth and therefore store the suffix link directly. Only one additional bit to mark a node as high or regular node is necessary. The complete layout of the nodes is shown in Figure 3.
The leaves simply store the starting position of the corresponding suffix in the text. To distinguish a pointer to a leaf from a pointer to an internal node, each reference has to be marked with one bit.
The suffix tree representation by Kurtz needs $S_{\text{Kurtz}} = 20\,\text{B}$ in the worst case and in practice 10.1 B for many real world texts [Kur99].
The representation can (due to the limited space for the pointers) be used for texts up to maximal length $2^{27} - 1 = 134\,217\,727$ [Kur99].

2.2.3.2 Construction

The suffix tree by Kurtz can be constructed using a linear-time algorithm, such as the algorithm of McCreight [McC76]. The worst case construction time is therefore $T^{\text{construct}}_{\text{Kurtz}} = \mathcal{O}(n)$.

2.2.3.3 Search

Searching a pattern of length m can be accomplished by simply following the child pointers (and iterating over the siblings). This gives asymptotically optimal time $T^{\text{bool}}_{\text{Kurtz}} = \mathcal{O}(m)$ and $T^{\text{count/pos}}_{\text{Kurtz}} = \mathcal{O}(m + occ)$.

Internal node:

5	27	2	1	29
dist	first child ...	first child	rightmost	next sibling / parent suffix link

Additionally for a large internal node of type *regular node*:

1	4	27	5	27
high		depth		head position

Additionally for a large internal node of type *high node*:

1	21	10	5	27
high	suffix link ...	depth	suffix link	head position

Leaf:

32
starting position

Figure 3: Suffix tree: node layout of the space reduced suffix tree by Kurtz.

2.2.3.4 Discussion

The suffix tree representation stores the information in the nodes very compactly, reducing the space requirements. It works, however, only up to a maximal text length of $2^{27} - 1$ and does not work well together with external memory (due to the necessary random accesses).
One advantage in some applications might be the existence of suffix links, which are missing in several other proposed suffix tree representation.

2.2.4 WOTD suffix tree

The *WOTD suffix tree*[5] is a representation and construction algorithm for suffix trees proposed by Giegerich et al. [Gie+03]. It is also called *lazy suffix tree* because the tree can be built in a lazy fashion, i. e., the nodes are only created when they are traversed for the first time.
The algorithm to build the tree is known under the name *WOTD*: It is called *write-only* because once a node has been evaluated it does not need to be touched again during the building process (unlike the classical algorithms for suffix tree construction where nodes are accessed again later). It is called *top-down* because the nodes are created starting from the root, and every time a node is created, its parent has already been created before.
The representation of the original proposal [Gie+03] is also called *STTD32*[6] to distinguish it from *STTD64* [Hal+07] which uses basically the same construction algorithm and is discussed in the following Section 2.2.5.
The WOTD suffix tree was already part of the SeqAn software library, implemented by Weese [Wee12], and the corresponding index class is called `IndexWotd`.

[5]WOTD = **w**rite-**o**nly, **t**op-**d**own
[6]STTD32 = **S**uffix **T**ree **T**op **D**own **32**-bit

2.2.4.1 Data structure

The representation of a WOTD suffix tree consists of two tables: the *suffixes* and the *nodes* array. The *suffixes* array is structurally the same as a suffix array, i. e., it is a table storing the starting positions of the text suffixes. However, the entries are initially not sorted in lexicographic order, but the sorting is done incrementally during the construction of the tree.[7]

The *nodes* array contains the nodes of the tree and is filled incrementally as the tree is constructed. There are two types of nodes: *evaluated* nodes and *unevaluated* nodes. For an easier understanding, we first describe the data structure of the completely constructed tree, i. e., where all nodes are evaluated.

A concept central to the representation of nodes is the *left pointer lp* of a node. It points to a starting position of the edge label in the text and is defined as follows: For a node $\overline{rs}$ that is a child of node $\overline{r}$ and has the incoming edge label s, the left pointer is $lp(\overline{rs}) :=$ $\min\{$ starting position of rs in the text $\} + |r|$. The left pointer therefore points at the first character of an occurrence of the edge label in the text.

The representation of each evaluated node stores the left pointer of the node (Figure 4). It additionally stores a bit marking it as internal node or leaf. The children of a node are stored consecutively in the *nodes* array. It is therefore not necessary to store a pointer to the next sibling, but it suffices to have an extra bit marking a node as rightmost child. Each internal node additionally stores a pointer to the first child in the *nodes* arrays, and a bit marking the node as evaluated or unevaluated (leaves are always evaluated).

The children of a node are stored in order of increasing left pointer, i. e., the child with the longest suffix in its subtree is the first child. The length of the incoming edge does not have to be stored explicitly because it can be calculated by the difference of the left pointer of the node and the left pointer of its first child (short proof in [Gie+03]).

The node layout in the implementation with a word size of 32 bit is depicted in Figure 4. In the implementation in SeqAn, a node is stored in a word of type `size_t`, which on a 32-bit machine usually equals an integer of 32 bit but can, for example on a 64-bit machine, also be larger. The additional field *\$-edges* is introduced in the implementation, leading to 30 bit for the first child or right pointer (instead of 31 bit as in the original proposal [Gie+03]).

During the construction of the tree, the *nodes* array also includes unevaluated nodes. These do not yet store the left pointer and the pointer to the first child, but are only placeholders and instead store borders of an interval of the *suffixes* array. The construction algorithm and the handling of unevaluated nodes is explained in more detail in the following subsection.

The *nodes* array occupies in the worst case $3n$ words of 32 bit (n words for the leaves and $2n$ words for the internal nodes); the size of the *suffixes* array is n words. The total space consumption is therefore $16\,n\,\mathrm{B}$ in the worst case (assuming a text character is stored in 1 B and the word size is 4 B = 32 bit). (In the original proposal by Giegerich et al. [Gie+03], the *suffixes* array is discarded once the complete tree is constructed, leading to a total space of $12\,n\,\mathrm{B}$.)

For the analysis of the construction memory we distinguish between the lazy and eager version:

- When using the *eager version*, the constructed tree needs $S_{\text{WOTD (eager)}} = 16\,n\,\mathrm{B}$. During the construction, additional working memory is needed for a temporary array in the counting sort of size n words $= 4\,n\,\mathrm{B}$ in the worst case. We furthermore need to maintain the unevaluated nodes in a stack of size $\frac{n}{2}$ entries in the worst case; each entry in the implementation consists of 24 B[8], so the stack in total needs $12\,n\,\mathrm{B}$ resulting in a total working memory of $S^{\text{construct}}_{\text{WOTD (eager)}} = (16 + 4 + 12)\,n\,\mathrm{B} = 32\,n\,\mathrm{B}$ for the construction.

[7]We therefore call it *suffixes array* and not *suffix array*.

[8]In the original proposal, the stack is smaller leading to a lower construction memory.

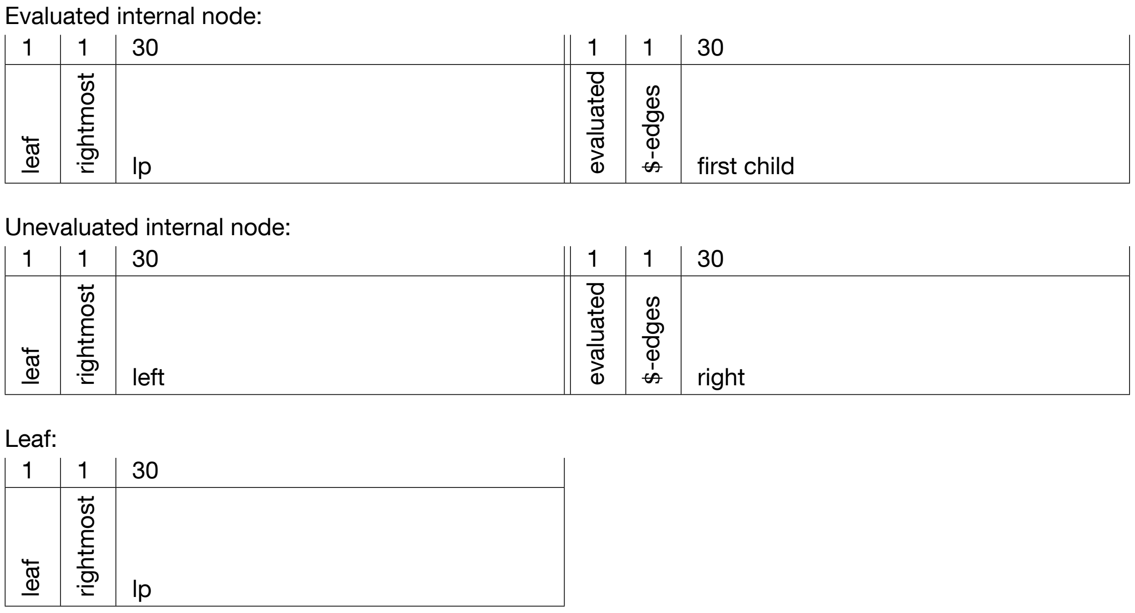


Figure 4: WOTD suffix tree: node layout.

- When using the *lazy version*, we need all data structures also during the usage of the tree since the construction is done simultaneously (except for the stack of unevaluated nodes which is not necessary here). This results in a total space usage of $S_{\text{WOTD (lazy)}} = (16 + 4)\,n\,\text{B} = 20\,n\,\text{B}$.

The text length is limited by the size of the pointers. Since the left pointers have 30 bit available, the text can not be longer than 2^{30} characters. However, the first child pointer (referencing the *nodes* array of size $3\ n$) is stored in 30 bit as well, yielding $3n < 2^{30} \Rightarrow n < 357\,913\,941$ for the completely built tree. (The first child pointer is stored in 31 bit in the original proposal leading to twice the possible text length [Gie+03].)

Index components of `IndexWotd`:

1. `WotdSA`: The *suffixes* array.
2. `WotdDir`: The *nodes* array.

Index parameters of `IndexWotd`:

1. `TIndexStringSpec`: This parameter chooses an implementation for the storage of the index tables, e. g., whether the tables should be held in memory or in secondary memory. Possible choices are the string types, e. g., `Alloc`, `External<TConfig>`, or `MMap<TConfig>`.
   (This parameter can not be passed directly, but has to be set by overloading a meta-function, e. g., `DefaultIndexStringSpec`.)

2.2.4.2 Construction

To construct the tree, the *nodes* array is filled up step by step. Initially it only contains the unevaluated root node. When an unevaluated node gets evaluated, new child nodes are created and appended to the *nodes* array. Following the recursive structure of a suffix tree, the construction algorithm processes recursively. The construction is shown in Figure 5.

To *evaluate* a node $\bar{r}$ means to create the child nodes of $\bar{r}$, which can be internal nodes and leaves. To do so it is necessary to determine the outgoing edge labels of $\bar{r}$. This is possible if we have available the set of remaining suffixes S of the subtree rooted at $\bar{r}$, where $S(\bar{r}) := \{\, s \mid rs \text{ is a suffix of } t \,\}$. The algorithm maintains the sets of remaining suffixes in the *suffixes* array, containing their starting positions. Each internal node corresponds to an interval of the *suffixes* array, the root node for example corresponds to the complete array.[9] The array initially contains the starting positions of all text suffixes (i. e., it contains the numbers 0 to $n+1$) and is resorted incrementally. The algorithm maintains the following invariant: For each unevaluated internal node $\bar{r}$, the interval $[\mathit{left}(\bar{r}), \mathit{right}(\bar{r})]$ of the *suffixes* array contains the starting positions of the remaining suffixes $S(\bar{r})$ in descending order of their length.

When an unevaluated internal node $\bar{r}$ is evaluated, the remaining suffixes $S(\bar{r})$ are sorted by the first character. This is done by sorting the interval $[\mathit{left}(\bar{r}), \mathit{right}(\bar{r})]$ of the *suffixes* array using counting sort and a temporary array. This yields for each character of the alphabet one (possibly empty) group of suffixes. These groups enable us to create the child nodes of $\bar{r}$:

- For an empty group, nothing has to be done.
- For a group of size 1, a new leaf is created.
- For a group of size ≥ 2, a new unevaluated internal node is created.

The suffixes are iterated in descending order of their length, and the new nodes are created in that order. This ensures a correct ordering of the children of a node (see [Gie+03]). Setting the attributes of the just evaluated node and of the newly created nodes goes straight-forward.

When a new unevaluated internal node is created, we have to ensure that the corresponding interval in the *suffixes* array contains the starting positions of the *remaining* suffixes of this node. Therefore all entries in the interval are increased by h where h is the LCP of all suffixes in the group. The LCP is computed in a simple loop, incrementing the currently found LCP by 1 and checking the corresponding character of all suffixes of the group in each step.

In the *lazy* version of this index, all nodes remain unevaluated until they are accessed for the first time. Only constant work $T^{\text{construct}}_{\text{WOTD (lazy)}} = \mathcal{O}(1)$ has to be done initially. The implementation contained the algorithm for lazy construction. The implementation additionally contains a mechanism to construct the *first level* of the tree, which is needed for every access to the tree nodes.

In the *eager* version of this data structure, the nodes are created using an additional stack of unevaluated nodes (even though the order of evaluation could in principle be variable). The eager construction algorithm takes $T^{\text{construct}}_{\text{WOTD (eager)}} = \mathcal{O}\left(n^2\right)$ quadratic time in the worst case, because $\mathcal{O}(n)$ nodes are evaluated, each taking $\mathcal{O}(n)$ time. In the expected case the construction needs $\mathcal{O}(n \log n)$ time [Gie+03]. (We implemented the eager construction by using the lazy construction algorithm and a simple depth-first search.)

Construction parameters of `IndexWotd`:

1. The variant for building the tree: `lazy` (default), `first`, or `eager`.

[9] The connection between suffix tree nodes and intervals of the *suffixes array* is similar to the classical *suffix array* (Section 2.1.1).

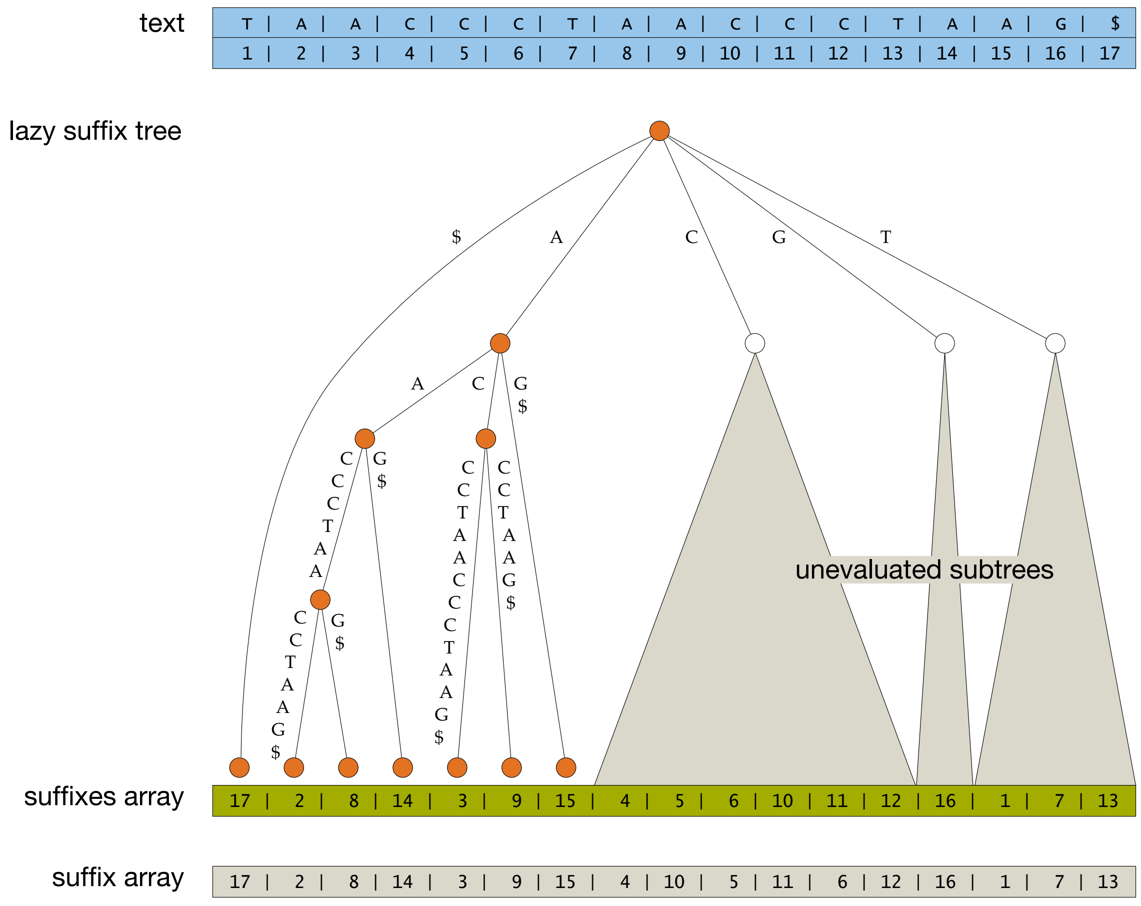


Figure 5: WOTD suffix tree: construction algorithm.

The WOTD suffix tree is constructed incrementally. The filled nodes are already evaluated, i. e., their children are created. The white nodes are unevaluated; the corresponding intervals of the *suffixes* array are therefore not sorted yet (compared to the suffix array which is shown for comparison).

2.2.4.3 Search

To find the occurrences of a pattern in a text using the WOTD suffix tree, we can use the generic algorithm for suffix trees and traverse the tree from the root. However, if we encounter an unevaluated node, we have to evaluate it first. As usual we process the pattern from left to right and descend in the tree. To compute the label of an edge, we use the infix of the text starting at the position indicated by the left pointer and ending at the position indicated by the left pointer of the first child minus 1.

When the algorithm successfully reaches the end of the pattern and stops in an internal node, we have to output all positions in the leaves of the subtree rooted at the current node. Therefore we have to perform a depth-first-search (keeping track of the current depth in the tree) and output for each leaf $\bar{r}$ the $lp(\bar{r}) - depth(\bar{r})$, since this is the starting position of the current text suffix.

Performing exact pattern matching of a pattern with length m in a text of length n using a lazy WOTD suffix tree needs $T^{\text{bool}}_{\text{WOTD (lazy)}} = \mathcal{O}(n\,m)$ and $T^{\text{count/pos}}_{\text{WOTD (lazy)}} = \mathcal{O}(n\,m + occ)$ time in the worst case, because at most m nodes need to be evaluated. However, once a node has been created it only takes constant time to traverse it. Therefore the lazy suffix tree performs much better in practice than this pessimistic asymptotic time bound suggests.

Performing exact pattern matching in a completely constructed WOTD suffix tree (e. g., with eager construction) takes $T^{\text{bool}}_{\text{WOTD (eager)}} = \mathcal{O}(m)$ and $T^{\text{count/pos}}_{\text{WOTD (eager)}} = \mathcal{O}(m + occ)$ worst case time.

2.2.4.4 Discussion

The WOTD suffix tree is especially interesting because not the whole suffix tree has to be built, which would only pay off if many search queries have to be answered. It works particularly well if only the upper part is accessed (and therefore only the upper part needs to be constructed) [Wee12].
A disadvantage of the WOTD suffix tree is a relatively slow search time for short patterns when we are interested in all matches (R_{count} or R_{pos}): If the traversal ends in a node close to the root, we have to traverse a huge subtree with bad locality of reference as noted by Halachev et al. [Hal+07]. This is improved with the STTD64 variant described in the following section.
A drawback for some applications are the missing suffix links, but for pattern matching they are not needed.
In the original proposal by Giegerich et al. [Gie+03], the nodes are stored in arbitrary order (depending on the construction algorithm). The implementation is modified so that the nodes are stored in lexicographic order.

2.2.5 STTD64 suffix tree

The suffix tree representation *STTD64* by Halachev et al. [Hal+07] is a modification of the representation of the WOTD suffix tree described above, which is called *STTD32* to distinguish them. STTD64 uses a node layout which occupies 64 bit for internal nodes and also for the leaves. The construction algorithm is extended with a partitioning step to be able to construct the suffix trees for even larger texts. The result is not one suffix tree, but one tree for each partition. The algorithm only works for the eager version of the construction.
Our implementation in the SeqAn software library has mainly been done during a student's project by Aumann [Aum11], the corresponding index class is called `IndexSttd64`.

2.2.5.1 Data structure

The STTD64 representation of the suffix tree again consists of the *nodes* array but the layout of the nodes is different compared to STTD32, see Figure 6. An evaluated internal node has a similar layout, except that we do not store $-edges and do not have to store the evaluated bit because the algorithm only works in the eager version and we keep track of the unevaluated nodes in a stack. This gives us full 32 bit for the left pointer referencing the text positions. The modification for unevaluated nodes goes analogously. The most substantial change for STTD64 is that the leaves also occupy 2 machine words = 64 bit, making it possible to additionally store the string depth inside each leaf (the length of the path label).

The STTD64 representation requires $S_{\text{STTD64}} = 4n$ machine words = $16\,n$ B in the worst case ($2n$ machine words for the internal nodes and $2n$ machine words for the leaves). Once the tree is constructed, it is not necessary to store the *suffixes* array. During the construction we need the same additional working memory as STTD32 of $10\,n$ B, resulting in $S^{\text{construct}}_{\text{STTD64}} = 26\,n$ B.
In an extended version of the construction algorithm (see below), the text suffixes are partitioned, so that two suffixes are in the same partition if and only if they have a common prefix of length *prefixLength* $\in \mathbb{N}$. The trees are constructed for each partition independently, the resulting data structure is then a set of trees. (Note that not the text itself is partitioned, but the suffixes are distributed to partitions based on their respective prefix.)
The maximal text length is limited to 2^{32} by the size of the left pointers. The first child pointer of evaluated nodes as well as the left and right fields of unevaluated nodes are restricted to 30 bit. However, these fields only refer to values within one partition. If the parameter *prefixLength* and

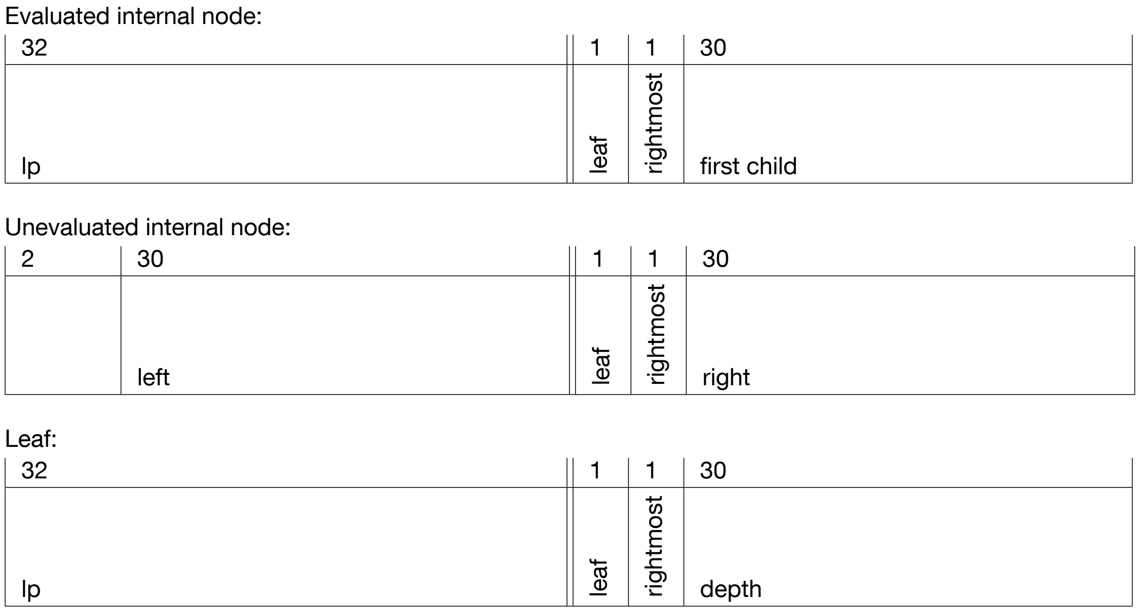


Figure 6: STTD64 suffix tree: node layout.

therefore the number of partitions is big enough, texts of length up to 2^{32} can theoretically be indexed with STTD64.

In our implementation by Aumann [Aum11] we added a small additional tree data structure to be able to traverse the virtual global suffix tree represented by the set of trees. This data structure is called *virtual prefix tree* and is a trie containing all text infixes of size *prefixLength*. Additionally it also contains the last few suffixes of the text, i. e., the suffixes $t_{[n-prefixLength+1\,..]}, t_{[n-prefixLength+2\,..]}, \ldots, t_{[n\,..]}$, which are too short to belong to any of the partitions. The size of the virtual prefix tree is $\sigma^{prefixLength}$, which is constant if we assume the alphabet size and prefix length to be constant (the prefix length is limited to values between 0 and 7 in practice).

Index components of `IndexSttd64`:

1. `Sttd64SuffixTrees`: The set of trees, each consisting of a set of nodes, stored in external memory as `String<External>`.
2. `Sttd64Partitions`: The set of partitions, only needed during the construction.
3. `Sttd64VirtualTree`: The virtual prefix tree.

Index parameters of `IndexSttd64`:

1. `inMem` (`bool`): Chooses between a variant optimized for main memory or external memory (default: `true`).
2. `PREFIX_LENGTH`: Length of the common prefix shared by suffixes in one partition ($0 \leq$ `PREFIX_LENGTH` ≤ 7). The resulting number of partitions is $\sigma^{\texttt{PREFIX\_LENGTH}}$.

2.2.5.2 Construction

The construction algorithm of the STTD64 representation by Halachev et al. [Hal+07] is an extension of the eager construction algorithm of the WOTD suffix tree (see previous Section 2.2.4). The unevaluated nodes are maintained in a stack, leading to a depth-first-search traversal during the evaluation of the nodes.
In our implementation, buffering techniques based on those proposed by Tian et al. [Tia+05] are used for the *nodes* array, the *suffixes* array and the temporary array in the counting sort. They are realized by using the external string provided by SeqAn which implements a *least recently used* strategy (LRU) to swap out pages. The buffering can be configured using parameters for the page size and the number of frame to be simultaneously kept in main memory.

A more efficient way to calculate the LCP has been incorporated in our implementation by Aumann [Aum11]: Instead of increasing the LCP counter in each step only by 1, the LCP counter is doubled in each step, leading to less iterations and resulting in a better performance in practice.

Partitioning. To build the suffix trees for large texts, the suffixes of the text are partitioned so that two suffixes are in the same partition if and only if they have a common prefix of length $\geq$ *prefixLength*. This is done be scanning the text and sorting the suffixes into the partitions based on their first *prefixLength* characters. Afterwards the tree is built for each partition independently. The virtual prefix tree is also built during the partitioning phase. The general scheme of the construction is shown in Figure 7.

One optional last step of the construction is to fill in the depth values of the leaf nodes. These fields have not been set during the creation or evaluation of the nodes since the values are not easily available at this time. To fill in the values, we finally simply traverse each tree in a depth-first-search fashion, keeping track of the current depth and storing it in the leaves. This step is executed by default at the end of the construction in our implementation.

The algorithm to build the STTD64 tree representation is an extension of the eager algorithm for the WOTD suffix tree. The partitioning and depth-filling only take additional linear time. Therefore the asymptotic running time does not change and the construction takes $T_{\text{STTD64}}^{\text{construct}} = \mathcal{O}(n^2)$ in the worst case.

Construction parameters of `IndexSttd64`:

1. `PAGESIZE`: The page size for the buffers, counted in bytes.
2. `TREEBUFF_FRAME_NUM`: The maximum number of frames that should reside in main memory for the *nodes* array. Should be at least 2 to perform well in situations at the border of pages.
3. `SUFBUFF_FRAME_NUM`: The maximum number of frames that should reside in main memory for the *suffixes* array. Should be at least the size of the alphabet σ because during the counting sort σ many positions are accessed in the worst case.
4. `TEMPBUFF_FRAME_NUM`: The maximum number of frames that should reside in main memory for the temporary array used in the counting sort.

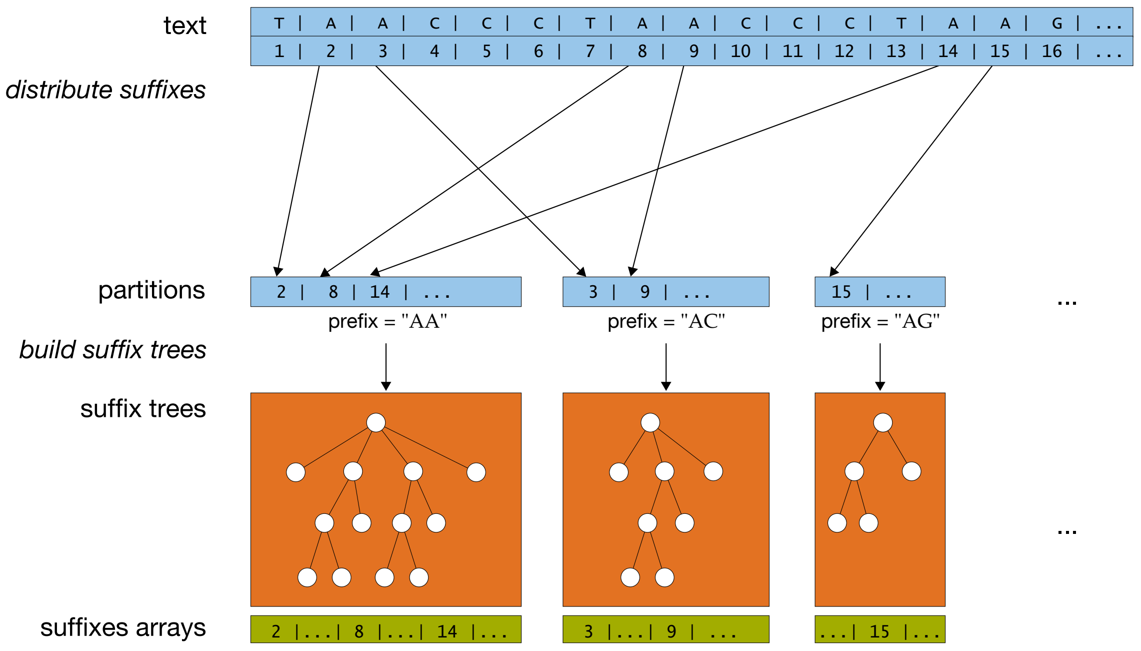


Figure 7: STTD64 suffix tree: construction algorithm using the optional partitioning.

The text suffixes are distributed to partitions according to their prefix. The trees of the partitions are then built independently.

2.2.5.3 Search

To find a search pattern in the text using the STTD64 representation, we first have to determine the correct partition. Therefore we use the small auxiliary virtual prefix tree which gives us the number of the partition to search in. (In the special case that the pattern is shorter than *prefixLength*, we simply have to output all leaves of the corresponding partitions.)
In the tree corresponding to the found partition, we descend from the root as usual and (in the case of a successful search) eventually end up in a node. Now we can apply an optimization compared to the STTD32 representation. For STTD32 we had to explicitly traverse all nodes of the subtree rooted at the current node and output for each leaf the position calculated from the left pointer and the current depth. For STTD64 we stored the depth inside each leaf and can therefore omit the traversal but instead iterate linearly over the nodes of the subtree. We therefore descend to the first child and go right in the *nodes* array until we found as many nodes with rightmost bit = 1 as internal nodes (then we have visited all nodes of the subtree). This works because the nodes have been created in depth-first-search order.
Performing exact pattern matching of a pattern with length m in a text using a STTD64 representation needs $T^{\text{bool}}_{\text{STTD64}} = \mathcal{O}(m)$ and $T^{\text{count/pos}}_{\text{STTD64}} = \mathcal{O}(m + occ)$ optimal time in the worst case, independent of the size of the underlying text.

2.2.5.4 Discussion

The STTD64 representation is only available with an eager construction algorithm and therefore more useful if many searches are carried out. Because of the partitioning, it can be built for longer texts than the STTD32 representation. Therefore, however, several parameters need to be tuned. Furthermore, searches of *all matches* of a given pattern (R_{count} and R_{pos}) can be answered faster, especially for short patterns with many matches because it circumvents the traversal of very big subtrees (which is necessary in the STTD32 representation).

2.2.6 Enhanced suffix array

The *enhanced suffix array* proposed by Abouelhoda et al. [Abo+04] consists of a suffix array (Section 2.1.1) with some additional tables and can be used to simulate a suffix tree.[10] It has the same asymptotic space consumption $\mathcal{O}(n)$, but needs less space compared to other representation (especially during the construction) and furthermore exhibits a good locality of reference.

The central idea of the enhanced suffix array is based on the correspondence between the suffix tree and the suffix array: Each node of the suffix tree corresponds to an interval of the suffix array (note: this statement does not hold the other way around). The nodes are therefore not stored explicitly, but are maintained at runtime as left and right border of intervals of the suffix array. When traversing from one node to another in the simulated suffix tree, we need a way to compute the new interval borders.

The enhanced suffix array therefore consists of the several additional tables as described below. The different tables are needed for different kinds of tree traversals (e. g., bottom-up, top-down) and can be built on demand. In the description here we focus on a top-down traversal in the simulated suffix tree because this is needed for pattern matching.

An implementation of the enhanced suffix array was already contained in the SeqAn software library, implemented by Weese [Wee12], and the corresponding index class is called `IndexEsa`. The problem instance can be a text or a set of strings which is implicitly concatenated to form a text as well.

2.2.6.1 Data structure

The table *SA* (called *suftab* in the original paper) is the suffix array of length n and the central component of the enhanced suffix array. It stores the starting positions of the suffixes of the text in lexicographic order as described in Section 2.1.1. The table *SA* requires $4\,n\,\mathrm{B}$ (one machine word per input character).

The table *lcptab* of length n stores the length of the LCP of two consecutive entries in the suffix array (see Section 2.1.2). Together with the *childtab* it can also be used to determine the depth of a node (the LCP of the entries in an interval of the suffix array) in constant time [Abo+04]. The *lcptab* requires $4\,n\,\mathrm{B}$.

The table *childtab* of length n is used to represent the parent-child relation of the nodes (i. e., of the suffix array intervals). For each such interval $[i, j]$, the borders of the first child and the borders of the next sibling are conceptually stored. A first idea could be to store the information for such an interval in *childtab*$[i]$. This, however, is not sufficient because several intervals can have the same left border i (e. g., a node and its first child). Therefore the following redundant scheme is proposed, conceptually consisting of three fields: *up*, *down*, and *nextlIndex*. The field *childtab*$[i]$.*up* or the field *childtab*$[j+1]$.*down* stores the right border of the first child, and the field *childtab*$[i]$.*nextlIndex* contains the information for the next sibling. Because many of these fields would be empty (since not every index marks the beginning or the end of an interval), it is possible to store all three values in only one field (we omit the details here). Constant time checks can be performed at runtime to determine which type of field is actually stored [Abo+04]. The *childtab* requires $4\,n\,\mathrm{B}$.

The suffix array together with the LCP table and the child table requires $S_{\mathrm{ESA}} = 12\,n\,\mathrm{B}$.

[10] Since we use it as virtual suffix tree, it is presented in this section and not together with the suffix array (Section 2.1).

Index components of `IndexEsa`:

1. `EsaSA`: The suffix array, realized as `String<SAValue, TIndexStringSpec>` where `SAValue` is for a text by default defined as `unsigned int`.
2. `EsaLcp`: The *lcptab*, realized as `String<TSize, TIndexStringSpec>`, where `TSize` is by default defined to be an `unsigned int`.
3. `EsaChildtab`: The *childtab* is a `String<TSize, TIndexStringSpec>`.
4. `EsaBwt`: The Burrows-Wheeler transform *bwttab*, realized as `String<TValue, TIndexStringSpec>`, where `TValue` is the alphabet type of the underlying text.
5. `EsaRawText`: If the problem instance is a text, this is simply a reference to the text. If the problem instance is a string set, this is a virtual concatenation of all the strings in the set.

Index parameters of `IndexEsa` (enhanced suffix array):

1. `TIndexStringSpec`: This parameter chooses an implementation for the storage of the index tables, e. g., whether the tables should be held in main memory or in secondary memory. Possible choices are the string types, e. g., `Alloc`, `External<TConfig>`, or `MMap<TConfig>`, see Section 1.6.
   (This parameter can not be passed directly, but has to be set by overloading a meta-function, e. g., `DefaultIndexStringSpec`.)

Space reductions. The authors propose two simple modifications for the practical implementation, reducing the space requirements of both *lcptab* and *childtab* from 4 n B to 1 n B each [Abo+04].
The entries in *lcptab* only occupy 1 B, limiting the maximal value to 255. Values greater than 255 are stored as 255 and additionally in a variable sized table, which is at runtime accessed using a binary search. This worsens the theoretical asymptotic running time, but is reported to work well in practice [Abo+04].
The entries in *childtab* also only occupy 1 B, limiting the maximal value to 255 as well. Values greater than 255 are stored as 255 only. When finding such a value, the correct value has to be calculated at runtime, worsening the theoretical asymptotic running time.
With these two modifications, the total space consumption of the enhanced suffix array is $S_{\mathrm{ESA}} = 6\,n$ B. These space reductions are, however, not included in the implementation.

Additional tables. The table *bwttab* of length n is not necessary for a top-down traversal of the tree, but can be used for a bottom-up traversal and other algorithms. It stores the Burrows-Wheeler transform (BWT) of the underlying text [BW94], which is a rearrangement/ permutation of the text characters. It is formally defined as: $bwttab[i] := t_{[SA[i]-1]}$, except when $SA[i] = 1$ then $bwttab[i] = t_{[n]} = \$$ [NM07].[11] This table can also be used to avoid random memory access to characters of the text. It is stored in 1 n B.
The additional table *suflink* of length n is proposed to store suffix links. However, this table is not included in the implementation because it is not needed for many algorithms including pattern matching.

[11] Despite the simple definition, the BWT is a very powerful tool, and Adjeroh et al. [Adj+08] devote a complete book to it.

2.2.6.2 Construction

To build the enhanced suffix array, the suffix array is constructed first by using one of the available algorithms (see Section 2.1.1) and then the LCP table is built using the algorithm of Kasai et al. [Kas+01] (see Section 2.1.2). Both can be achieved in $\mathcal{O}(n)$ worst case time. Then the child table can be built by linearly scanning the LCP table maintaining a stack [Abo+04]. The total time to build the enhanced suffix array is therefore $T_{\text{ESA}}^{\text{construct}} = \mathcal{O}(n)$ in the worst case, using $S_{\text{ESA}}^{\text{construct}} = \mathcal{O}(n)$ working memory.
The implementation contains modified algorithms adapted for the use in external memory [Wee12].

Construction parameters of `IndexEsa` (enhanced suffix array):

1. `TAlgSpec`: Algorithm to create the suffix array (see Table 1 on page 21).
2. (Algorithm specific parameters if applicable.)

2.2.6.3 Search

Solving the exact pattern matching problem with the help of the enhanced suffix array can be solved with no asymptotic slowdown compared to a classical suffix tree representation. The search starts at the virtual root node (corresponding to the interval $[1, n]$) and processes the pattern as usual from left to right. A character of the pattern is read and the correct child interval is determined by using the table *childtab*, which asymptotically needs $\mathcal{O}(\sigma)$ steps in the worst case. Then the depth of the current node is determined by using the table *lcptab*. The corresponding number of characters of the search pattern is compared to the text using the starting position stored in the suffix array. If the comparison was successful, the search continues with the next character of the search pattern. Finally, when reaching the end of the pattern, all positions in the current interval of the suffix array are returned.
Performing exact pattern matching of a pattern with length m in a text using an enhanced suffix array needs $T_{\text{ESA}}^{\text{bool/count}} = \mathcal{O}(m)$ and $T_{\text{ESA}}^{\text{pos}} = \mathcal{O}(m + occ)$ optimal time in the worst case, independent of the size of the underlying text.

2.2.6.4 Discussion

The enhanced suffix array is relatively small compared to traditional linked suffix tree representation. The construction algorithms in the implementation are adapted for the use in external memory. However, to perform pattern matching (i. e., to traverse the tree top-down) in external memory, several random memory accesses are necessary.
A unified presentation of enhanced suffix arrays can be found in the book by Ohlebusch [Ohl13].

2.2.7 Suffix forest in external memory

The standard algorithms to build a suffix tree (like Ukkonen's algorithm, see Section 2.2.2) make heavy use of random memory accesses. This is no problem as long as the suffix tree fits into main memory, but the performance degrades considerably when the tree is bigger so that it has to be stored in external memory. This, for example, is the case for the DNA sequence of the human genome (about 3 billion base pairs), where the suffix tree representation by Kurtz [Kur99] (see Section 2.2.3) would need about 30 GB, not fitting into the main memory of a regular desktop computer. However, not only the performance of the construction but also of the search algorithm suffers when the classical suffix tree representations are stored in external memory.

Internal node:

32	32	32	32
left child	right child	leftmost leaf	depth

Leaf:

32
starting position

Figure 8: DiGeST index: node layout.

This is because each step from one node to another involves a random memory access, which in the worst case requires an I/O operation in an external memory setting.
Therefore some similar representations have been proposed that use a data structure resembling intuitively speaking a *suffix forest*, i. e., a collection of *treeCount* $\in \mathbb{N}$ small partial suffix trees. The partial trees are stored on disk and the size of one tree is such it can be quickly loaded into main memory. The basic idea to perform an exact search of a pattern is then as follows: The algorithm uses some information available in main memory to determine which partial tree to access, loads this tree from external memory, and performs the search in the partial suffix tree.
Representations of this kind are used, for example, in *Trellis*[12] by Phoophakdee and Zaki [PZ07], its successor *Trellis+* by the same authors [PZ08], *DiGeST*[13] by Barsky et al. [Bar+08]. A survey by Barsky et al. [Bar+10] describes and experimentally compares several different approaches for the construction of suffix trees or forests in external memory. The best performance among the tested index structures and algorithms has DiGeST, especially for large inputs. An implementation is already provided by the original authors, but it operates on files, has no clean programming interface, is restricted to DNA sequences, and is not very flexible regarding choices of parameters. Therefore we chose to implement this index structure to use it in our experimental comparison. In the implementation we follow the original proposal, but extend the index structure and algorithm, so that it is also possible to use it in more general applications (as compared to being restricted to DNA sequences only) and with more flexible parameter settings. The implementation has been done by Dau and Krugel, the corresponding index class is called `IndexDigest`.

2.2.7.1 Data structure

To simplify the data structure and to save space, the DiGeST index is not built on the text itself, but instead on a binary representation of the text. Each node of the partial suffix trees then has exactly two children, one for the character 0 and one for the character 1. In the original proposal, the encoding is limited to the DNA alphabet $\{A, C, G, T\}$; we extended this here to arbitrary alphabets so that it is also possible to use, for example, a natural language text.
A *partial suffix tree* is a compacted trie for suffixes in an interval of the suffix array of the underlying text. Each partial tree contains an instance of the root node (which represents the empty string) and all paths that lead to suffixes of the corresponding suffix array interval. To represent a partial suffix tree we store the set of nodes, the set of leaves (which is an interval of the global suffix array), and the offset of this interval within the global suffix array. Each node stores links to both its children, the index of the leftmost leaf of the corresponding subtree, and the depth of the node

[12]Trellis is an anagram of the bold letters in the phrase: **E**xternal **S**uffix **TR**ee with Suffix **L**inks for **I**ndexing Genome-sca**L**e Sequences.
[13]DiGeST = **Di**sk-Based **Ge**nomic **S**uffix **T**ree

(length of the path from the root node, counted in bit). The resulting node layout with a word size of 32 bit is depicted in Figure 8.
All partial suffix trees have the same size (plus/minus 1 node), leading to a better memory consumption and I/O behavior as compared to other approaches. (Similar proposals like Trellis [PZ07] require that all suffixes stored in one partial suffix tree share a common prefix, leading to an unbalanced distribution of tree sizes.) The size of the partial suffix trees can in our implementation be controlled using the parameter `OUTBUF_SIZE` (counted in terms of internal nodes), which should by sufficiently small that a tree can efficiently be loaded from secondary to main memory.
To be able to search in the suffix forest, we also need to store some information telling us which lexicographic interval of suffixes is stored in which partial suffix tree. This can be achieved by maintaining an array *dividers* of *treeCount* = $\lceil 2n / \texttt{OUTBUF\_SIZE} \rceil$ elements in main memory, so that the ith divider represents the lexicographically greatest suffix stored in the ith partial suffix tree. Instead of storing the complete suffix explicitly, only a fixed sized *binary prefix* is stored, together with the starting position of the suffix in the text. This makes it possible to store *dividers* in main memory and therefore to access it without an additional I/O operation. The size of this binary prefix can be controlled using the parameter `PREFIX_LENGTH` (counted in bit).

The DiGeST representation requires $S_{\text{DiGeST}} = 9n$ machine words = $36\,n\,\text{B}$ in the worst case ($2 \cdot 4n$ machine words for the internal nodes and n machine words for the leaves). For each of the *treeCount* partial trees an additional constant amount of space is needed to store the offset and entry in *dividers*. Because this additional space is comparatively very small (only a few bytes for each partial tree), it is neglected in the analysis of the space usage here.

Index components of `IndexDigest`:

1. `DigestSuffixTrees`: Collection of partial suffix trees (stored in external memory as `String<External>`), each of which consists of:
   - the set of nodes,
   - the leaves (an interval of the global suffix array), and
   - the offset of this interval within the global suffix array.
2. `DigestDividers`: Sorted array of *dividers* (stored as `String<Alloc>` in main memory).

Index parameters of `IndexDigest`:

1. `OUTBUF_SIZE`: Size of one partial suffix tree (measured in terms of internal nodes). The space usage of one partial tree is `OUTBUF_SIZE` $\cdot$ 18 B.
2. `PREFIX_LENGTH`: Length of the binary text prefixes stored in *dividers* and used for merging the suffix array entries (measured in bit).

2.2.7.2 Construction

The algorithm for constructing the DiGeST index by Barsky et al. [Bar+08] and in our implementation works in three phases: First it splits the text into smaller partitions, then builds the suffix array for each partition in main memory, and finally merges the suffix arrays to build and output the partial suffix trees to disk. This is visualized in Figure 9.

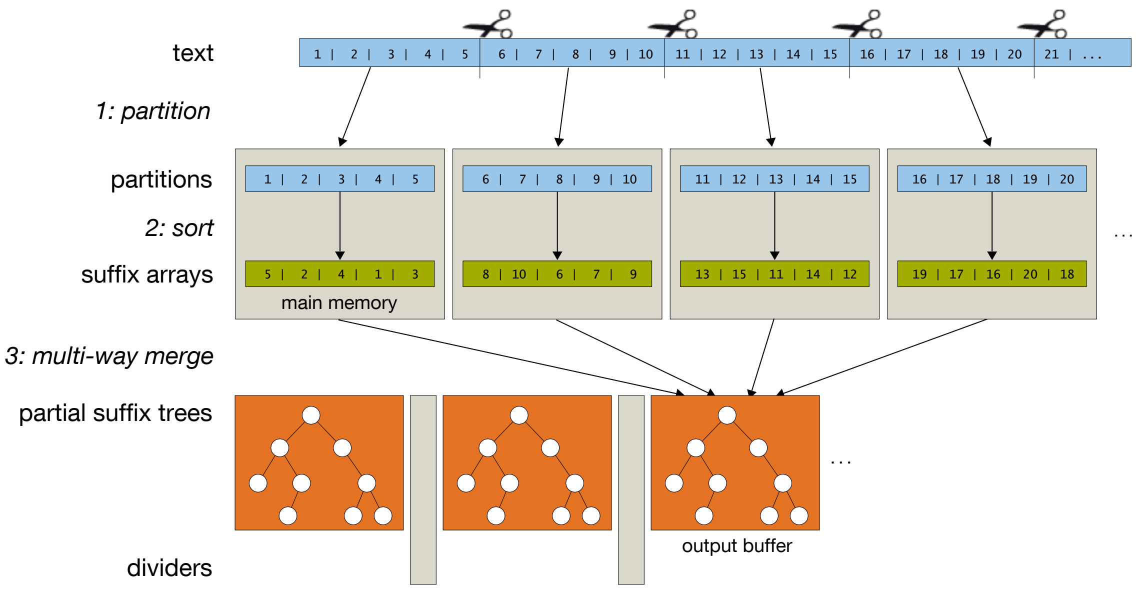


Figure 9: DiGeST index: construction algorithm in three phases.
The text is split in partitions of fixed size, the suffix array for each partition is built in main memory and the suffixes of all partitions are merged into a set of partial suffix trees. Additionally, the lexicographically greatest suffix of each tree is stored in main memory as an array of dividers.

1st phase. In the partitioning phase, the text is split into $|partitions| = \lceil n / \texttt{PARTITION\_SIZE} \rceil$ partitions of equal size (except for the last partition which can be shorter). In the implementation, the size of the partitions can be controlled using the parameter `PARTITION_SIZE`, which should be chosen so that it is possible to construct the suffix array for one partition in main memory. Additionally, the algorithm appends to each partition (except the last) a prefix of the next partition, called *tail*. This is necessary to guarantee a correct sorting: If the tail of a partition does not occur as substring inside the partition, then the sorting of the suffixes is correct, also with respect to the *suffix array of the whole text* (short proof in [Bar+08]). In the implementation, the length of the tail can be controlled using the parameter `TAIL_LENGTH`.[14]
The output of the partitioning phase is a list of starting and end positions of the partitions. (Note that the DiGeST index partitions the text itself and does not distribute the text suffixes as the STTD64 construction algorithm.)

2nd phase. In the second phase, the algorithm iterates over the partitions and builds the suffix array for each partition, including the tail. This is done in main memory and it is possible to use any available construction algorithm for suffix arrays (see Section 2.1.1). In the implementation, the algorithm can be chosen using the template parameter `TAlgorithm`. After one suffix array has been built, it is stored in external memory (entries referring to text positions in the tail are filtered out).
In addition to each text position, we also store a binary prefix of the suffix starting at this position. This is useful later when merging the suffix array and helps avoiding most random accesses to the text. The length of this binary prefix can be controlled with the parameter `PREFIX_LENGTH`. The binary representation of the prefix is stored in a consecutive array of machine words (in practice this can be, for example, 64 bit, stored in two unsigned integers). The output of this phase is a set of suffix arrays and each entry additionally contains the short text prefix.

[14]It has to be noted that the DiGeST index does not guarantee to correctly build the index if there are repetitions in the text of length > `TAIL_LENGTH`.

3rd phase. In the third phase, the suffix arrays are merged. To process the entries in global lexicographic order, we use a priority queue containing the smallest entry of each suffix array. After an entry of one suffix array has been processed, the next entry of this suffix array is inserted into the queue. To reduce I/O operations we maintain for each suffix array an input buffer, whose size can be controlled with the parameter `INBUF_SIZE`.
The suffix entries in the priority queue are ordered lexicographically. To speed up the comparisons, we use a user-defined compare function that takes into account the stored binary prefixes instead of always comparing directly in the underlying text. This takes advantage of the locality of reference and avoids random memory access, which would even lead to I/O operations if the text is stored in external memory. The compare function is implemented to use efficient comparisons of machine words and bit operations, and only falls back on comparing in the text if the binary prefixes are equal.
The partial suffix trees are constructed using an output buffer of predefined size, which in the implementation is controlled by the parameter `OUTBUF_SIZE`. Once the buffer is full, the current tree is written to disk and a new empty tree is initialized. One partial tree consists of the set of internal nodes and an interval of the global suffix array which represents the leaves of the partial tree. To insert a suffix into a partial suffix tree, the algorithm follows the rightmost path from the root node up to the depth of the LCP (here counted in bit) with the previously inserted suffix.[15] In the standard case, a new internal node is created. (In the special case that one suffix is a prefix of another, no internal node is created.) Afterwards a new entry is added to the set of leaves.

The time to construct the DiGeST suffix forest is $T^{\text{construct}}_{\text{DiGeST}} = \mathcal{O}\left(n^2\right)$ in the worst case: Phase 1 and Phase 2 take each $\mathcal{O}\left(n^2\right)$ time if a linear time algorithm for suffix array construction is used. Phase 3 requires $\mathcal{O}\left(n^2\right)$ time in the worst, because n suffixes are inserted into the output buffer and each insertion needs $\mathcal{O}(n)$ time in the worst case. This total running time, however, is quite pessimistic, and a theoretical average time of $T^{\text{construct}}_{\text{DiGeST}} = \mathcal{O}(n \log n)$ is given by Barsky et al. [Bar+10] (under certain assumptions, e. g., that the characters of the text are independently and randomly distributed).
The additional space usage during the construction $S^{\text{construct}}_{\text{DiGeST}}$ depends on the suffix array construction algorithm, $\lceil n / \texttt{PARTITION\_SIZE} \rceil$ suffix arrays are created and stored in external memory, each of length `PARTITION_SIZE`. The parameters of the construction should be chosen such that each construction can be carried out within main memory.

Construction parameters of `IndexDigest`:

1. `TAlgorithm`: Suffix array construction algorithm (see Section 2.1.1, default: `LarssonSadakane`)
2. `PARTITION_SIZE`: Size of the partitions of the input text.
3. `TAIL_LENGTH`: Length of the tail which is virtually appended to each partition for sorting.
4. `INBUF_SIZE`: Size of the input buffer of each suffix array during the merging phase (measured in array entries consisting of one integer for the starting position and the stored binary prefix).

[15] In the original proposal, Barsky et al. [Bar+08] describe a binary search on the rightmost path in the tree. This seems to be not included in the implementation provided by the authors and we also use a simple linear search.

2.2.7.3 Search

Exact pattern matching using a DiGeST index goes straightforward. The algorithm first generates the binary representation of the search pattern and locates the greatest divider which is smaller. We implemented a binary search in *dividers* and use the stored binary prefixes of the dividers for the comparisons. A comparison can therefore in many cases be carried out without any random memory access to the text. Only if the stored binary prefix of a divider is equal to the prefix of the search pattern, we need to resort to comparing the further characters of the underlying text.
When the correct divider has been found, the corresponding partial suffix tree is loaded from external to main memory. Starting at the root node, we go down the tree using the skip/count trick described in Section 2.2.1 until we eventually reach a leaf node or have processed the whole search pattern. All leaves in the subtree of the current node are potential matches. Because we blindly followed the edges in the tree using the skip/count trick, we need to verify whether it is an actual match. Therefore we simply check for one of the potentially matching positions, whether it is a match by resorting to the underlying text. We only have to verify one of the suffixes because by construction they all share the same prefix. We use the lexicographically smallest position which is available by using the pointer to the leftmost leaf of the current node. If the verification was successful, the algorithm returns all text positions in the subtree of the current node.
In the original proposal [Bar+08], the case of a pattern occurring in more than one tree is not described (which happens for example if the search pattern is very short). To handle this case in our implementation, the algorithm continues to compare the search pattern with the next divider, loads the next partial suffix tree if necessary etc.
Binary searching in *dividers* takes $\mathcal{O}(\log \mathit{treeCount} \cdot m)$ worst case time. Searching in one of the trees takes $\mathcal{O}(m)$ time, yielding a total worst case search time of $T^{\text{bool}}_{\text{DiGeST}} = \mathcal{O}(\log \mathit{treeCount} \cdot m)$ and $T^{\text{count/pos}}_{\text{DiGeST}} = \mathcal{O}(\log \mathit{treeCount} \cdot m + \mathit{occ})$.

2.2.7.4 Discussion

Suffix forests in external memory in general and the DiGeST representation in particular allow for an efficient construction, even if the resulting index does not fit into main memory. It furthermore allows to answer exact pattern matching queries with *only one access* to external memory for loading a small partial tree (*dividers* and the text have to be kept in main memory).
One disadvantage is that the index occupies much space (which, however, can be stored in external memory). The resulting data structure is not a real suffix tree and therefore does not permit all operations on suffix trees, such as different kinds of traversal or using suffix links.

The array *dividers* actually resembles a *sampled suffix array* because it maintains the starting position of text suffixes in lexicographical order, but stores only some sampled entries (≈ multiples of `OUTBUF_SIZE`/2).[16] The other, not sampled entries of the suffix array are stored in the leaves of the partial trees. It seems that this connection has not been mentioned in the literature before. The parameter `OUTBUF_SIZE` can then be seen as a way to choose between a data structure more similar to a suffix array (for small values) or more similar to a single suffix tree (for greater values).

2.2.8 Other suffix trees in external memory

There are several algorithms for constructing suffix-trees (or suffix-tree-like data structures) in external memory: Hunt et al. [Hun+01] propose an external memory index and algorithm for the use in biological databases, and Schürmann and Stoye [SS03] extend this algorithm.

[16]Sampled suffix arrays are used in several compressed index structures such as the FM index (Section 2.3.1) and the compressed suffix array (Section 2.3.2).

Other approaches are the *Distributed and Paged Suffix Tree* by Clifford and Sergot [CS03], the *Top-compressed suffix tree* by Japp [Jap04], the *TDD (Top Down Disk-based)* algorithm based on WOTD (Section 2.2.4) by Tian et al. [Tia+05], the *Stellar layout* for suffix trees by Bedathur and Haritsa [BH05], an DNA index based on the binary representation by Won et al. [Won+06], the *CPS-tree (Compact Partitioned Suffix tree)* by Wong et al. [Won+07], *Trellis* and *Trellis+* by Phoophakdee and Zaki [PZ07] and Phoophakdee and Zaki [PZ08], *B^2ST* by Barsky et al. [Bar+09; Bar+11b], and the *ERA (Elastic Range)* algorithm by Mansour et al. [Man+11].

2.3 Compressed indexes

The index structures presented in the previous sections all occupy space that is several times the size of the underlying text (the factor for suffix arrays is 4 and for suffix tree representations in the order of 20). Therefore these index structures do not fit into main memory even for moderately sized texts. Instead of finding ways to efficiently use the indexes in external memory, much research has been carried out to compress the indexes. This is based in the observation of Donald E. Knuth that "*space optimization is closely related to time optimization in a disk memory*" (in *The Art of Computer Programming* [Knu98, Section 6.5], [FN05]). If a compressed index fits into main memory it can still be faster than an index in external memory, even if some calculations have to be performed to decompress the stored information.

The suffix array needs several times the space to store the underlying text. When measuring the space in bit, one simple observation is as follows. A text with n characters from a finite alphabet of size σ can be stored in $\Theta(n \log \sigma)$ bit space: each of the n characters can be encoded by $\Theta(\log \sigma)$ bit. The corresponding suffix array stores a permutation of $\{1, \ldots, n\}$ and therefore occupies $\Theta(n \log n)$ bit space: each of the n entries stores the position of a suffix in $\Theta(\log n)$ bit. Therefore the suffix array needs asymptotically more space than the text with a factor of $\log_\sigma n$. Not all permutations of $\{1, \ldots, n\}$ are a valid suffix array and therefore it should be possible to store the suffix array with less space. From an information theoretical point of view, suffix arrays can be represented by $\Theta(n \log \sigma)$ bit, since they are in correspondence with a text of n symbols. However, this observation does not help much to find a succinct representation of a suffix array because a lookup would take $\Omega(n)$ time if the suffix array was represented by the underlying text itself. The goal for compressed suffix arrays is therefore to derive a compact representation that still permits to answer lookup queries efficiently.
Since not every permutation of $\{1, \ldots, n\}$ represents a suffix array and since natural language texts usually exhibit regularities (e. g., repeated substrings), also the entries of suffix arrays show some regularities (e. g., so-called *self-repetitions*, i. e., regions that occur again only shifted by a constant offset) [NM07].

There are many different compressed index structures and variants. They use various compression techniques: sampling, run-length encoding, differential encoding, wavelet-trees, and others. Due to the compression, the size of the indexes often depends on the empirical entropy H_g of order $g \in \mathbb{N}$ of the underlying text (the entropy measure is described in Section 5.2.4). The size is therefore usually measured in bit (instead of B as for the index structures in the other sections) [NM07]. It is desirable that the size of the index grows with H_0 or even H_g (with $g > 0$) because this means that the resulting index is smaller if the text is compressible.
Several compressed index structures can be used to even *replace the text* (so that the text can be discarded after construction) by making it possible to reconstruct the text from the information stored in the index. Such an index is called a *self-index* [NM07].
Compressed indexes usually support bidirectionality, i. e., it is possible to extend the pattern in both directions while searching. In a suffix tree it is, however, only possible to add characters

at the end of a string to refine the search. Because many compressed indexes support this bidirectionality, they are in this sense more powerful, which is exploited by some algorithms for approximate pattern matching (Section 3.3.7) [Rus+09a; Ohl13].

Navarro and Mäkinen [NM07] present the ideas, concepts, and compression techniques, and furthermore give an extensive overview of compressed index structures and their theoretical background.
Ferragina et al. [Fer+09a] focus on the practical aspects of several compressed index structures, discuss implementation details, and give an experimental comparison. They furthermore introduce the *Pizza & Chili* website that provides implementations of several popular indexes, and also test data to experimentally evaluate the implementations [FN05].

Three of the most popular compressed index structures are implemented in the software library SeqAn and described in the following sections, namely the FM index, the compressed suffix array (CSA), and the LZ index.

2.3.1 FM index

The *FM index*[17] is a popular family of compressed full-text indexes and based on the original proposal by Ferragina and Manzini [FM00; FM05]. Several variants of the basic data structure are discussed by Navarro and Mäkinen [NM07].
The FM index (abbreviated as *FMI*) is interesting here because it indexes a text in such a way that the resulting data structure has a size in the order of the size of the text (or even of the compressed text) and still permits to efficiently answer pattern matching queries. The FM index is based on the Burrows-Wheeler transform (BWT) of the underlying text [BW94; NM07] (which can also be used with the enhanced suffix array and described in Section 2.2.6.1 on page 41).
A version of the FM index based on the variant by Ferragina et al. [Fer+04] is implemented by Singer [Sin12] in the software library and is available under the name `FMIndex`.

2.3.1.1 Data structure

The index structure stores the BWT of the underlying text in a way that permits to efficiently answer so-called *rank* queries (also called *occurrences queries*) efficiently, since this is the only operation needed on the BWT when performing pattern matching in the FM index. A rank query can formally be written as $rank_u(i)$ and means "*How often does character u occur in the BWT up to position i?*" [NM07; Fer+09a].
There are different ways of storing the necessary information of the BWT. In the implementation, this information is called *occurrence table* and it is possible to choose between different implementations: rank support bit strings (called *sequence bit mask*) [FM00] and a wavelet tree [Fer+04] as described by Navarro and Mäkinen [NM07]. The first variant is reportedly better for small, the latter for larger alphabets.
The FM index furthermore contains a *sampled suffix array* where (in contrast to the classical suffix array) only some entries are stored explicitly (a fraction of 1 / `compressionFactor` where `compressionFactor` $\in \mathbb{N}$ is a parameter of the index). The remaining entries have to be computed during the search if necessary.
Additionally, the FM index contains a small table C of size σ that stores for each character $u \in \Sigma$ the number of occurrences of text characters which are smaller than u. This information together with the occurrences table is sufficient to *count* the matches of a pattern in the text.
Details about the used data structures in the implementation are described by Singer [Sin12].

[17] FM = **F**ull-text index in **M**inute space

The overall space consumption of the FM index depends on the empirical entropy of the text (for a definition see Section 5.2.4) and is $S_{\text{FMI}} = \mathcal{O}\left(H_g n + n \log \sigma\right)$ bit (the small constant factors and the order of the entropy depend on the concrete implementation, especially on the variant of the occurrences table and the sample rate) [NM07].

Index components of `FMIndex`:

1. `FibreLfTable`: Contains the table C together with the occurrences table and depends on the parameter `TOccTable`.
2. `FibreSA`: The sampled suffix array (with rate `compressionFactor`, see below).
3. `FibrePrefixSumTable`: The table C of length σ storing one value per alphabet character.

Index parameters of `FMIndex`:

1. `TOccSpec`: The type specialization for the occurrence table): `SBM` = sequence bit mask, `WT` = wavelet tree.
2. `compressionFactor`: Sample rate of the sampled suffix array (default: 10).

2.3.1.2 Construction

The construction algorithm of the FM index proceeds in several steps: First it builds the full suffix array (in the implementation this is done using the `Skew7` algorithm). Then the sampled suffix array is created, followed by the array C, and a temporary table storing the BWT. Finally, the occurrence table is built from the BWT. All steps require $\mathcal{O}(n)$ time, resulting in an overall construction time $T^{\text{construct}}_{\text{FMI}} = \mathcal{O}(n)$. The construction space $S^{\text{construct}}_{\text{FMI}}$ is dominated by the space to build the full suffix array (Section 2.1.1).

2.3.1.3 Search

To perform exact pattern matching of a pattern p of length m in a text, the FM index uses a method called *backward search*. The idea is to traverse the pattern from back to front (for $i = m$ down to $i = 0$) and to calculate in step i the interval in the suffix array that contains all text suffixes starting with $p_{[i\,..]}$. This interval can iteratively be refined using the previously calculated interval for $p_{[i+1\,..]}$, the stored information of the BWT, and the array C. A step takes constant time (when using the sequence bit mask variant) or $\mathcal{O}(\log \sigma)$ time (when using the wavelet tree), resulting in $T^{\text{bool/count}}_{\text{FMI (SBM)}} = \mathcal{O}(m)$ and $T^{\text{bool/count}}_{\text{FMI (WT)}} = \mathcal{O}(m \log \sigma)$. To also determine the positions of the matches (and not only the interval borders in the virtual suffix array), the sampled suffix array is used. Locating one match takes worst case $\mathcal{O}\left(\frac{\log \sigma \log^2 n}{\log \log n}\right)$ time, yielding $T^{\text{pos}}_{\text{FMI}} = T^{\text{count}}_{\text{FMI}} + occ \cdot \mathcal{O}\left(\frac{\log \sigma \log^2 n}{\log \log n}\right)$. A detailed description of the backward search algorithm is given by Navarro and Mäkinen [NM07]. The underlying text itself is not accessed when using backward search and therefore has not necessarily to be kept.

2.3.1.4 Discussion

The main advantage of the FM index compared to the classical index structures (such as the suffix array) is its compact size. Therefore the index might fit into main memory for texts where

other index structures already have to be swapped out to disk. The time to access the external memory can easily outweigh the time needed to decode the values.
The space depends on the empirical entropy of the text, which is desirable because it is smaller if the text is compressible. The asymptotic space bound is reported to be quite pessimistic [NM07].
The FM index is a self-index and can therefore also be used to extract characters of the text, while the text itself does not have to be kept after construction.
A disadvantage of the FM index is the relatively high space consumption during the construction because the uncompressed suffix array is built first. There are approaches reducing the construction space by Hon et al. [Hon+03; Hon+07a] building a compressed version of the suffix array first.
The idea of the FM index and the BWT on a high level is described by Willets [Wil03]. Practical aspects and a comparison with the compressed suffix array (next section) are described by Hon et al. [Hon+04] and for larger texts by Hon et al. [Hon+10]. A variant optimized for repetitive texts (e. g., genomes) was proposed by Sirén et al. [Sir+09].
Bowtie by Langmead et al. [Lan+09] and Bowtie 2 by Langmead and Salzberg [LS12] are efficient open-source implementations of the FM index for bioinformatics, in particular for read alignment in genomes. FEMTO[18] by Ferguson [Fer12] is a recent implementation of the FM index for external memory and was successfully used for texts of more than 100 GB.

2.3.2 Compressed suffix array

The *compressed suffix array (CSA)* by Sadakane [Sad00] and Sadakane and Shibuya [SS01] is also called *Sad-CSA* to distinguish it from other compressed suffix arrays [NM07]. It is based on the compressed suffix array *GV-CSA* by Grossi and Vitter [GV00; GV05], but modified and turned into a self-index (so that the text is not needed after construction).
Our implementation in the SeqAn software library has been done during a student's project by Stadler [Sta11], the corresponding index class is called `IndexSadakane`.

2.3.2.1 Data structure

The data structure of the CSA consists of the following four components [SS01]:

Ψ: The core component is the function $\Psi(i)$ that returns for an entry of the suffix array the position of the next shorter suffix: $SA[\Psi(i)] = SA[i] + 1$. It is formally defined as:

$$\Psi(i) := \begin{cases} SA^{-1}[1] & SA[i] = n \\ SA^{-1}[SA[i] + 1] & \text{otherwise} \end{cases}$$

SA denotes the suffix array and SA^{-1} the inverse suffix array (see Section 2.1.1). The use of this function becomes clearer when describing the search algorithm.

Due to the structure of the suffix array, the function Ψ is piecewise monotonously increasing and there are at most σ such pieces (see proof in [NM07, Lemma 1]). This helps to store a compressed representation of Ψ.

The function Ψ is stored explicitly only at positions that are multiples of the input parameter `PSI_SAMPLE_RATE` $\in \mathbb{N}$. The other values are stored using a differential encoding by Elias [Eli75].

To calculate the value of Ψ at an arbitrary position, the next smaller explicitly stored value is used as starting point and the differential encoding provides the necessary remaining information.

[18] FEMTO = **F**M-index for **e**xternal **m**emory with **t**hroughput **o**ptimizations

I: The suffix array is represented by table I. However, the values are stored explicitly only at positions that are multiples of the parameter SA_SAMPLE_RATE $\in \mathbb{N}$. The other values can be calculated using the function Ψ and a simple loop in $\mathcal{O}(\log n)$ time [Sad03].

J: Additionally also the inverse suffix array is included and represented by table J. The values are stored explicitly only at positions that are multiples of the parameter ISA_SAMPLE_RATE $\in \mathbb{N}$. The other values can be calculated using the function Ψ and a simple loop in $\mathcal{O}(\log n)$ time [Sad03].

C: Another small table C of size σ stores for each character $u \in \Sigma$ the number of occurrences of text characters smaller than u (just as in the FM index).

The overall space consumption of the compressed suffix array is $S_{\text{CSA}} = nH_1 + \mathcal{O}(n)$ bit $= \mathcal{O}(n)$ bit, where H_1 denotes the first order empirical entropy of the text [SS01].

Index components of `IndexSadakane`:

1. `FibreSA` is of type `SadakaneData` which contains
   (a) `psiSamples`: An array of type `TSize[]` storing the sampled values of Ψ.
   (b) `compressedPsi`: The differentially encoded values of function the Ψ are stored in a container for bits of type `TBitBin**`.
   (c) `I`: The sampled suffix array of type `TSize[]`.
   (d) `J`: The sampled inverse suffix array of type `TSize[]`.
   (e) `C`: The array C of type `TSize[]` and size σ.

Index parameters of `IndexSadakane`:

1. `PSI_SAMPLE_RATE`: The sample rate for Ψ (default: 128).
2. `SA_SAMPLE_RATE`: The sample rate for the suffix array I (default: 16).
3. `ISA_SAMPLE_RATE`: The sample rate for the inverse suffix array J (default: 16).

2.3.2.2 Construction

The compressed suffix array can be constructed *directly* (needing substantial additional working memory) or *incrementally* (needing only little additional space) [Hon+03].

To *directly* construct the compressed suffix array, the full suffix array is built first (see Section 2.1.1) and the inverse suffix array is calculated based on that. Afterwards, the data structures for Ψ, I, and J can be constructed. Finally, C is built by scanning the text. The overall running time of this construction algorithm is $T_{\text{CSA}}^{\text{construct direct}} = \mathcal{O}(n)$ (if a linear time algorithm to build the suffix array is used). The disadvantage of this direct method is that not only the compressed suffix array but also the uncompressed tables are needed during the construction. The total space needed during the construction is $S_{\text{CSA}}^{\text{construct direct}} = S_{\text{CSA}} + 2 \cdot S_{\text{SA}} = S_{\text{CSA}} + 8\,n\,\text{B}$.

To *incrementally* construct the compressed suffix array [Hon+03], the text can be split into $f = \lceil \frac{n}{h} \rceil$ partitions $t^1, \ldots, t^f$, where h is the length of each partition. The function Ψ is first calculated for the last partition t^f, then incrementally for $t^{f-1}t^f$, afterwards for $t^{f-2}t^{f-1}t^f$ etc. until it is calculated for the whole text. Finally, the arrays C, I and J are built. The overall running time of this construction algorithm is $T_{\text{CSA}}^{\text{construct incremental}} = \mathcal{O}(n \log n)$.

Construction parameters of `IndexSadakane`:

1. `blockLength`: The size f of the partition for the incremental construction.

2.3.2.3 Search

The original proposal uses the classical binary search (as described in Section 2.1.1) to find a pattern using the suffix array, here in its compressed representation [Sad03].
For the actual implementation, Sadakane and Shibuya [SS01] propose to use *backward search* just as for the FM index described in the previous Section 2.3.1 (and our CSA implementation also uses this approach). Again, an interval in the (virtual) full suffix array is maintained and iteratively refined as the pattern is traversed from back to front. A step from $p_{[i+1\,..]}$ to $p_{[i\,..]}$ uses the stored function Ψ and table C and takes $\mathcal{O}(\log n)$ time. This results in $T_{\text{CSA}}^{\text{bool/count}} = \mathcal{O}(m \log n)$. Locating the position of one match using the sampled suffix array takes $\mathcal{O}(\log n)$ worst case time, yielding $T_{\text{CSA}}^{\text{pos}} = \mathcal{O}(m \log n + occ \log n)$. The text is not accessed when using backward search and does therefore not necessarily have to be kept.

2.3.2.4 Discussion

The main advantage of the compressed suffix array compared to the classical index structures (such as the suffix array) is its compact size. Therefore the index might fit into main memory for texts where other index structures already have to be swapped out to disk. The time to access the external memory can outweigh the time needed to decode the values.
The compressed suffix array is a self-index so that it can also be used to extract characters of text, while the text itself does not have to be kept after construction.
Approaches reducing the construction space are proposed by Hon et al. [Hon+03; Hon+07a] and Sirén [Sir09].
A ''quick tour'' on compressed suffix arrays is given by Grossi [Gro11]. Many of the variants based on compressing the suffix array are described and compared by Navarro and Mäkinen [NM07]. Practical aspects and a comparison with the FM index (previous section) are described by Hon et al. [Hon+04] and for very large texts by Hon et al. [Hon+10].

2.3.3 LZ index

The *Lempel-Ziv index* (short: *LZ index* and *LZI*) is another family of compressed index structures that can be used for pattern matching. It is based on a Lempel-Ziv partitioning of the underlying text. The Lempel-Ziv compression methods represent a text by replacing a repeated substring by a pointer to a previous occurrence of this substring. There are several variants of LZ indexes, here we focus on the LZ index by Navarro [Nav02; Nav04], also called *Nav-LZI* [NM07]. This index structure is interesting for practical pattern matching applications since it is small and fast for position queries [Fer+09a]. Our implementation in the SeqAn software library has been done during a student's project by Stadler [Sta11], the corresponding index class is called `IndexLZ`.

2.3.3.1 Data structure

The LZ index by Navarro [Nav04] is based on the LZ78 algorithm by Ziv and Lempel [ZL78] that partitions a text t into h phrases $r_i \in \Sigma^*$ such that $t = r_1 \circ r_2 \circ \ldots \circ r_h$. Each phrase r_i is formed by appending one character to a previous phrase r_j, formally: $r_i = r_j \circ u$ with $j < i, u \in \Sigma$ and the additional artificial phrase $r_0 := \epsilon$.

The implementation of the LZ index consists of the following components:

1. *LZTrie* is a trie (see Section 2.2.1) storing the phrases $r_0, r_1, \ldots, r_h$. The trie has h nodes (plus the root node representing $r_0 = \epsilon$), since every phrase is formed by concatenating a single character to another phrase.

2. *LZRevTrie* is a trie storing the reversed phrases (each phrase read from right to left) $\overleftarrow{r_0}, \overleftarrow{r_1}, \ldots, \overleftarrow{r_h}$.

3. *Node* stores a mapping from a phrase number i to the node in *LZTrie* representing r_i. It can be used to traverse from a node in *LZRevTrie* to the corresponding node in *LZTrie*.

LZTrie allows to find all phrases starting with a given string, while *LZRevTrie* allows to find all phrases ending with a given string. The tries are stored in a space-efficient way using bit vectors with the so-called *parentheses representation* for the tree structure allowing constant time for tree traversal operations (such as navigating to the parent, child etc.), and an array of phrase identifiers [NM07].
The overall space requirement of the LZ index is $S_{\text{LZI}} = 4h \log h$ bit which can be bounded in terms of the empirical entropy of the text by $S_{\text{LZI}} = 4nH_g + o(n \log \sigma)$ bit for $g = o\left(\log_\sigma n\right)$ [NM07].
Instead of a two-dimensional range data structure to map nodes from *LZTrie* to *LZRevTrie* our implementation uses a mapping as described by Navarro and Mäkinen [NM07][p. 69] for practical implementations.
In addition to the original proposal, another comparatively small data structure is included in our implementation. This data structure stores sampled block starting positions to to be able to output absolute text positions, and not only phrase numbers (similar to what is described by Navarro [Nav09]).

Index components of `IndexLZ`:

1. `FibreSA` contains `LZData` which consists of
   (a) `LZTrie` (including the parentheses representation, the array of phrase identifiers, and the *Node* array) and
   (b) `LZRevTrie` (including the parentheses representation, and the array of phrase identifiers).

Index parameters of `IndexLZ`:

1. `LZTRIE_NODE_END_TEXT_POSTION_SAMPLE_RATE`: The sample rate for the additional data structure mapping phrases to absolute text positions (default: 128).

2.3.3.2 Construction

The construction algorithm of the LZ index is straight-forward: First, the LZ78 partitioning of the text into phrases $r_1, \ldots, r_h$ is computed. Then the phrases are inserted into *LZTrie* (at the same time, the mapping in the *Node* array is updated). Afterwards the reversed phrases are inserted into *LZRevTrie*.
Each insertion takes $\mathcal{O}(\sigma)$ worst case time, resulting in an overall construction time of $T_{\text{LZI}}^{\text{construct}} = \mathcal{O}(n \log \sigma)$ [AN11].

2.3.3.3 Search

When performing exact pattern matching of a search pattern p using the LZ index, we have to handle the following three distinct cases to find all matches [Nav04; NM07; Fer+09a]:

1. The pattern p is completely contained within one phrase r_i
2. The pattern p spans the border of two consecutive phrases r_i and r_{i+1}. A prefix $p_{[..x]}$ of the pattern is a suffix of r_i and a suffix $p_{[x+1..]}$ of the pattern is a prefix of r_{i+1} for some $x \in [1, m]$.
3. The pattern p is contained in the concatenation of at least three consecutive phrases $r_i, \ldots, r_j$. A prefix $p_{[..x]}$ of the pattern is a suffix of r_i, $p_{[x+1..y]} = r_{i+1} \circ \ldots \circ r_{j-1}$, and a suffix $p_{[y+1..]}$ of the pattern is a prefix of r_{i+1} for some $x, y \in [1, m]$.

The search algorithm checks for a given pattern all three cases and outputs all text positions corresponding to the found phrases [Nav04; NM07; Fer+09a], on a high level:

1. The algorithm searches all phrases ending with p using *LZRevTrie*, jumps to the corresponding node in *LZTrie* (using the *Node* array) and outputs all nodes in the subtree.
2. The algorithm has to try all possible split positions x of the pattern into $p_{[..x]}$ and $p_{[x+1..]}$. *LZRevTrie* is used to find the nodes corresponding to this pattern suffix and *LZTrie* is used for the pattern prefix. For each phrase r_i in the found subtree of *LZRevTrie* the algorithm checks whether r_{i+1} is contained in the found subtree of *LZTrie* and if so outputs the phrase.
3. If the pattern spans at least three phrases, some phrases are completely contained in the pattern. The algorithm essentially checks all $\mathcal{O}(m^2)$ substrings of the pattern and verifies whether the matching phrase can be extended (since all phrases are distinct, there can be at most one matching phrase).

The running times of the three cases are analyzed by Navarro and Mäkinen [NM07] as follows:

1. $\mathcal{O}(m^2 \log \sigma + occ_1)$
2. $\mathcal{O}(m^3 \log \sigma + (m + occ_2) \log n)$
3. $\mathcal{O}(m^2 \log \sigma + m^3)$

This gives the following overall worst case running time to find a pattern of length m in a text of length n using the LZ index: $T_{\mathrm{LZI}}^{\mathrm{count/pos}} = \mathcal{O}(m^3 \log \sigma + (m + occ) \log n)$. Counting the occurrences of a pattern using the LZ index is not faster than locating them. To answer only a boolean existence query the running time is essentially the same: $T_{\mathrm{LZI}}^{\mathrm{bool}} = \mathcal{O}(m^3 \log \sigma + m \log n)$.

2.3.3.4 Discussion

The LZ index is a self-index, i. e., it can also be used to extract pieces of the text so that the underlying text does not necessarily have to be stored.
The size of the index depends on the gth-order empirical entropy of the text, which is desirable because it is smaller if the text is compressible. However, the constants are quite big compared to the other compressed index structures discussed above.
For counting the occurrences of an exact pattern matching query it is rather slow in theory as well as in practice (especially for longer patterns). However, since locating the occurrences does not require extra work it is rather fast for finding the matching positions [Fer+09a].

There are alternative algorithms for a more space-efficient construction of the LZ index by Arroyuelo and Navarro [AN11] requiring asymptotically the same working memory as the final index. Other approaches reduce the space requirement of the index [Arr+06] and make it perform better in an external memory setting [AN07]. Navarro [Nav09] describe in detail many aspects of the implementation of the LZ index. There are several other variants of LZ based indexes described by Navarro and Mäkinen [NM07]. Russo and Oliveira [RO08] propose a compressed LZ index with an improved searching time.

2.3.4 Compressed suffix trees

Just as it was possible to extend the basic (uncompressed) suffix array with additional tables to provide suffix tree functionality (ESA, Section 2.2.6), it is also possible to add some compressed tables to a compressed index to form a *compressed suffix tree (CST)*. There are several different proposals by Grossi and Vitter [GV00; GV05], Sadakane [Sad07], Russo et al. [Rus+08a], Russo et al. [Rus+08b], Fischer et al. [Fis+08], Fischer et al. [Fis+09], Gog and Fischer [GF10], Sirén [Sir10], and Ohlebusch et al. [Ohl+10]. More recently, practical aspects of the implementation of compressed suffix trees are discussed by Välimäki et al. [Väl+09] (requiring in practice about 30 n bit space which is close to the size of a suffix array) and Cánovas and Navarro [CN10] (requiring in practice only about 8 to 16 n bit). Theoretical aspects of parallel and distributed compressed suffix arrays and trees as described by Russo et al. [Rus+10]. Since for a compressed suffix tree it is known that "*the implementation is by no means trivial and involves significant algorithm engineering*" [CN10], we did not implement this index structure ourselves.

2.4 *q*-gram indexes

Data structures using the q-grams of a text (defined below) are very popular indexes for pattern matching – on the one hand because they are quite simple (the idea as well as the implementation), and on the other hand because they perform well in practical applications.

The q-grams of a text are the consecutive substrings of length $q \in \mathbb{N}$.[19] In the literature, a q-gram is also called, e. g., *n-gram* or *k-mer*. The set of the q-grams of a string r is denoted by $Q_q(r) := \{\, r_{[i\,..\,i+q-1]} \mid i \in [1, |r| - q + 1] \,\}$; the 3-grams of the string r = "neighbor" are, e. g., $Q_3(r) = \{\,$"nei", "eig", "igh", "ghb", "hbo", "bor"$\,\}$.

The *positional q-grams* Q_q^{pos} of a string r are all pairs of text position and the q-gram starting at that position: $Q_q^{\text{pos}} := \{\, (i, r_{[i\,..\,i+q-1]}) \mid i \in [1, |r| - q + 1] \,\}$.

For the use in an index structure, the parameter q is usually small (e. g., often $3 \leq q \leq 10$) because the space consumption can grow exponentially with q.

There are many variants of q-gram based index structures. What they all have in common is that they store some kind of *inverted lists*: Instead of storing for each text position the q-gram starting at that position, the index stores for each q-gram at which text positions it occurs. This corresponds basically to an inversion of Q_q^{pos} from a mapping "text position $\rightarrow$ q-gram" to a mapping "q-gram $\rightarrow$ text position". The various index structures differ in how many and which q-grams are actually indexed and how the information is stored.

A q-gram index storing the q-grams of a text allows to efficiently perform exact pattern matching for patterns of length exactly q. The q-gram index can, however, also be used to search for *shorter and longer* patterns as described in the following sections.

[19] There are also *gapped* q-grams, i. e., q-grams that contain gaps. Here we focus on *ungapped* q-grams as they are used in most applications and are better suited for approximate pattern matching.

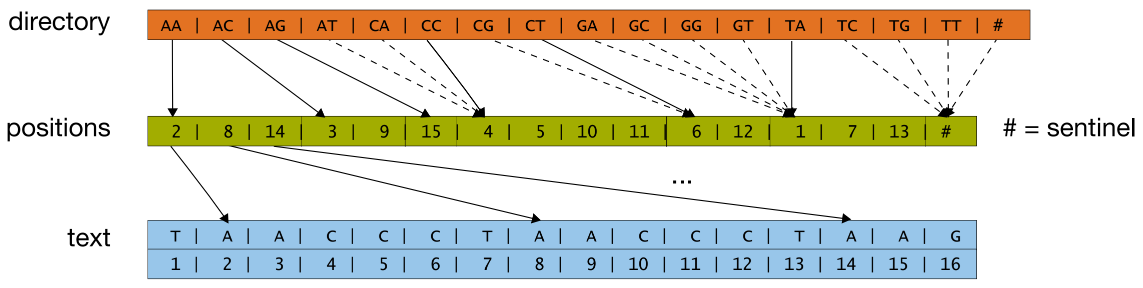


Figure 10: *q*-gram index: data structure.

The *q*-gram index (here for the text t = "TAACCCTAACCCTAAG" and $q = 2$) consists of the *directory* and the *positions* table. The *directory* can be accessed by using hash values of *q*-grams and points to the corresponding starting entry in the *positions* table. The *positions* table stores the actual starting positions in the text. (Figure based on [GDR09].)

2.4.1 *q*-gram index

The classical *q*-gram index stores the starting positions for *all* *q*-grams that are contained in the text (unlike the *q*-sample index described in the next section). An implementation was already contained in the software library and the corresponding index is called `IndexQGram`. A detailed description of the implementation is given by Weese [Wee12].

2.4.1.1 Data structure

The data structure conceptually consists of an array having one entry for each possible *q*-gram, and each entry points to a list of starting positions of this *q*-gram (possibly empty and usually sorted in ascending order).
There are many different possibilities on how to actually implement the data structure. The given implementation consists basically of two arrays: the *positions* table and the *directory* [GDR09] (Figure 10).

- The *position* table stores the starting positions of all *q*-grams of the text and is a permutation of the set $\{1, \ldots, n - q + 1\}$. The array is organized in σ^q blocks, where each (possibly empty) block contains the starting positions of one *q*-gram. The blocks are stored in lexicographic order of the corresponding *q*-gram, and the entries in each block are sorted in ascending order.[20]

- The *directory* stores for each possible *q*-gram r the starting position of the corresponding block in the *positions* table. (The end position of a block can be determined using the entry for the next greater *q*-gram.) The function *hash* : $\Sigma^q \rightarrow [1, \sigma^q]$ maps a *q*-gram to a natural number which is used to access the *directory*. It uses the function *ord* : $\Sigma \rightarrow \{0, \ldots, \sigma - 1\}$ that maps a character of the alphabet to a natural number.

The memory usage of this implementation of the classical *q*-gram index is as follows (when using a word size of 32 bit):

$$\begin{aligned} S_{q\text{-gram}} &= |directory| + |positions| \\ &= (\sigma^q + n) \cdot 4\,\mathrm{B} \end{aligned}$$

[20]Note the close connection between the *positions* table and the suffix array (Section 2.1.1): Both store permutations of starting positions of the text. In the suffix array the positions are ordered by the lexicographic order of the suffixes, and in the *positions* table they are ordered by the first *q* characters of the suffixes only. Due to the close connection, the *positions* table is called *SA* in the implementation [GDR09].

If the alphabet size is assumed to be constant, the space needed is $\mathcal{O}(n)$. However, due to the exponential dependence on the alphabet size, the memory usage of the *directory* cannot be neglected in practice.
The maximal text length is limited to 2^{32} by the size of the entries in the tables. However, the size of the *directory* can also be a limiting factor because its size σ^q is limited to 2^{64} (the hash values are stored in 64 bit integers). Depending on the alphabet size σ, the parameter q is limited. For the usual alphabet size $\sigma = 256$, q must not exceed 8 to have $\sigma^q \leq 2^{64}$. If the alphabet size, however, is only $\sigma = 4$ as for DNA sequences, q is allowed to take values up to 32, being sufficient for most practical applications. For wide characters (which can take 32 bit each, $\sigma = 2^{32}$), it has to hold $q \leq 2$.

Variant: Hashing with open addressing. To avoid the exponential space usage of the *directory*, it is possible to apply an alternative hashing method using open addressing. The size of the *directory* then depends on the text length, and another hashing function is used to map the q-grams to the buckets. The buckets are not sorted lexicographically with this variant. The additional table `QGramBucketMap` is used to store the mapping. To simplify the exposition we focus on the standard variant (also called *direct addressing* for distinction) in the following – the variant using open addressing needs at some points small modifications of the algorithms.

Index components of `IndexQGram`:

1. `QGramSA`: The *positions* table is a `String<SAValue, TIndexStringSpec>` where `SAValue` is for a text defined as `unsigned int`.
2. `QGramDir`: The *directory*, realized as `String<TSize, TIndexStringSpec>` where `TSize` is for a text defined as `unsigned int`.

Index parameters of `IndexQGram` (classic):

1. `Q`: The length of the q-grams.
2. `TSpec`: `OpenAddressing` or `void` (direct addressing, default)
3. `TIndexStringSpec`: This parameter chooses an implementation for the storage of the index tables, e. g., whether the tables should be held in memory or in secondary memory. Possible choices are the string types, e. g., `Alloc`, `External<TConfig>`, or `MMap<TConfig>`.
   (This parameter can not be passed directly, but has to be set by overloading a meta-function, e. g., `DefaultIndexStringSpec`.)

2.4.1.2 Construction

The classical q-gram index can be built using count sort in three simple steps as described by Gogol-Döring and Reinert [GDR09]:

1. Iterate over the text and compute the size of each bucket by counting for each q-gram the number of occurrences (using the *directory* as temporary storage).
2. Sum up the values in the *directory* table so that the entries serve as pointers to the starting positions of the blocks in the *positions* table.

3. Iterate over the text and insert each q-grams in its respective bucket, increasing the corresponding value in the *directory* by 1 after each insertion.

During the iterations over the text, we need to compute the hash values of consecutive q-grams. It is not necessary to compute the hash value anew for each q-gram (resulting in a running time of $\mathcal{O}(q)$ per step), but it is possible to use the hash value of the preceding q-gram and to compute the next value in constant time (independent of q) [GDR09].
The construction of the classical q-gram index takes $T^{\text{construct}}_{q\text{-gram}} = \mathcal{O}(n)$ time in the worst case.

2.4.1.3 Search

A q-gram index can be used to find a pattern p of length m in a text. We distinguish three cases and first describe the algorithm for $m = q$, then for $m < q$ and finally for $m > q$. In the implementation, pattern matching using a `Finder` and the q-gram index was restricted to $m = q$, we added the algorithms for the other two cases.
To find the occurrences of a pattern with length q (i. e., a q-gram) in a text using the classical q-gram index works as follows. The hash of the q-gram is computed in $\mathcal{O}(q)$ time and the corresponding entry in the *directory* is accessed. This value points to the left border of the occurrences in the *positions* table. The right border can be computed by subtracting 1 from the following entry in the *directory* (corresponding to the next greater q-gram in lexicographic order). With the left and right border at hand we can output all values in the *positions* table. For a counting query we only have to determine the size of the interval in the *positions* table.
It is also possible to efficiently find the occurrences of a pattern with $m < q$. The occurrences of such a p span across several q-grams in the *directory*, in particular all q-grams starting with p.[21] We can use basically the same algorithm to compute the hash value and use the *directory* to determine the left border in the positions array. However, the computation of the right border has to be modified slightly to compute the maximal hash value of a q-gram starting with p. Outputting the results works the same as above.
The q-gram index can also be used to find longer patterns with $m > q$. We implemented the following algorithm: One q-gram of the pattern is extracted and the occurrences in the text are searched using above algorithm. The matching candidate positions are then verified for an actual match of the whole search pattern. The q-gram to be extracted can in principle be any of the q-grams of the pattern. Our search algorithm uses the q-gram with the fewest entries in the *positions* table to reduce verification costs (based on Navarro and Baeza-Yates [NBY98]).

Exact pattern matching of a pattern p with limited length $m \leq q$ in a text using a classical q-gram index needs $T^{\text{bool/count}}_{q\text{-gram}} = \mathcal{O}(q)$ and $T^{\text{pos}}_{q\text{-gram}} = \mathcal{O}(q + occ)$ worst case time, independent of the size of the underlying text (which is optimal if $m = q$ or q is assumed to be constant).
Performing exact pattern matching of a pattern p with length $m > q$ in a text using a classical q-gram index needs $T^{\text{bool/count/pos}}_{q\text{-gram}} = \mathcal{O}(q + cand \cdot m) = \mathcal{O}(q + n\,m)$ time in the worst case, where *cand* denotes the number of matching candidate positions. This is, however, a rather pessimistic worst case bound and q-gram indexes are known to perform fast in practical settings (especially if the number of occurrences is high [Pug+06]).

Variant. This basic search algorithm can also be extended to extracting several q-grams and intersecting the respective entries of the *positions* table. This is used, e. g., by Puglisi et al. [Pug+06], Zobel and Moffat [ZM06], and Transier and Sanders [TS08; TS10].

[21] This algorithm for patterns shorter than q therefore only works if the q-grams are sorted lexicographically, i. e., with the direct addressing variant and not with open addressing.

2.4.1.4 Discussion

One advantage of the *q*-gram index compared to other index structures is its simplicity. The idea but also the implementation of the data structure and its algorithms are not very complicated.
The index structure can also be efficiently used in secondary memory. When searching a *q*-gram in the index, one random memory access to the *directory* is necessary and another to access the consecutive interval of the *positions* table. This is opposed to, e. g., the suffix tree, where the search of a pattern traverses the tree from the root, and each access to another node could involve a random memory access.
Furthermore, the underlying text is not needed when searching for *q*-grams because such queries can be answered using only the index. However, in the verification phase, we also have to perform comparisons in the text.
Another advantage in some applications is that when searching for patterns of length *q*, the found matching positions are sorted by increasing text position, which can be useful for further processing.

A disadvantage of the *q*-gram index is that it is restricted to rather simple operations on *q*-grams and does not allow all operations as suffix trees (such as tree traversal to find common substrings etc.).
Even though the *q*-gram index is often used and works well in practice, the worst case time bounds for performing pattern matching with patterns of arbitrary length are rather bad. The time to search a pattern depends on the structure of the text: If a *q*-gram of the pattern occurs more often in the text, the search takes longer (even if the pattern does not occur in the text at all).
The *q*-gram index is not very space-efficient because for each additional character of the text we have to store another entry in the *positions* table. In real-world applications, the text often contains repetitions which can be exploited: If a substring of the text occurs again in the text, the order of the *q*-grams within both substrings is the same and should therefore not be stored redundantly. This observation is used in the *q*-gram index with two levels to save some space (Section 2.4.3).

Approximate pattern matching in a *q*-gram index is usually implemented by using partitioning into exact search (Section 3.3.3) [ZD95; Li+08].

2.4.2 *q*-sample index

The *q*-sample index is a simple variant of the classical *q*-gram index designed to save space. It is a sampled *q*-gram index: instead of indexing the *q*-grams at every text position, only a subset of the positions is sampled and included in the index.[22] This idea is, for example, used by Sutinen and Tarhio [ST96] and Navarro et al. [Nav+00; Nav+05].
In the implementation of the software library it is possible to define a fixed step size *stepSize* to sample the *q*-grams. The classical *q*-gram index can then be seen as a special case with *stepSize* = 1.

2.4.2.1 Data structure

The data structure of the *q*-sample index is structurally the same as for the *q*-gram index. The difference is that the *positions* table does not contain all starting positions but only the sampled positions. The space consumption compared to the classical *q*-gram index is therefore partly decreased by a factor of *stepSize* as follows:

[22]The *q*-grams included in the index are called *q-samples*.

$$\begin{aligned} S_{q\text{-sample}} &= |directory| + |positions| \\ &= \left(\sigma^q + \left\lceil \frac{n}{stepSize} \right\rceil\right) \cdot 4\,\mathrm{B} \end{aligned}$$

Index parameters of `IndexQGram` (sampled):

1. `Q`: The length of the *q*-grams.
2. `TSpec`: `OpenAddressing` or `void` (direct addressing, default)
3. `TIndexStringSpec`: (see classical *q*-gram index above, Section 2.4.1)
4. `setStepSize()`: Sets the sample rate *stepSize*. When used for exact pattern matching, it should hold $m \geq q + stepSize - 1 \Leftrightarrow stepSize \leq m - q + 1$ to guarantee finding all matches.

2.4.2.2 Construction

The construction of the *q*-sample index works basically the same as for the *q*-gram index. However, instead of iterating over all text positions, the algorithm jumps forward by *stepSize* in each step. (Therefore it is in general not possible to use the hash value of the preceding *q*-gram to calculate the next hash value as described for the *q*-gram index construction above.) The construction time is $T^{\text{construct}}_{q\text{-sample}} = \mathcal{O}(n\,/\,stepSize)$.

2.4.2.3 Search

Since not every position of the text is indexed in the *q*-sample index, it is not possible to efficiently find *all occurrences* of a given *q*-gram in the text. However, we can use the *q*-sample index to efficiently locate the occurrences of a sufficiently long pattern. For a pattern *p* of length $m = q + stepSize - 1$ the algorithm proceeds as follows: Extract all *q*-grams of the pattern (there are $stepSize - 1$ many) and look up their positions in the index. Verify each found occurrence by comparing the text at this position with the pattern.
We extended this algorithm for longer patterns (just as for the *q*-gram index) by choosing among several possible extracted *q*-grams the one with the fewest entries in the *positions* table to reduce verification costs.
The *q*-sample index imposes a restriction on the minimal pattern length: For patterns of length $m < q + stepSize - 1$, not all occurrences can in general be found because an occurrence might not be indexed by a *q*-sample. Therefore the parameter *stepSize* should be chosen sufficiently small depending on the expected pattern lengths.
Performing exact pattern matching of a pattern *p* with length $m \geq q + stepSize - 1$ in a text using a *q*-sample index takes $T^{\text{bool/count/pos}}_{q\text{-sample}} = \mathcal{O}(q \cdot stepSize + cand \cdot m) = \mathcal{O}(q \cdot stepSize + n\,m)$ time in the worst case where *cand* denotes the number of matching candidate positions.

2.4.2.4 Discussion

The *q*-sample index is a simple approach to effectively decrease the space consumption of the *positions* table by a factor of *stepSize*. This comes at the cost of an increased search time.
Navarro et al. [Nav+00; Nav+05] also describe an algorithm for *approximate* pattern matching in a *q*-sample index.

2.4.3 q-gram index with two levels

An approach to reduce the space consumption while still indexing every q-gram without losing any information, is a two level data structure called *n-Gram/2L* by Kim et al. [Kim+05].[23] It is based on the following two observations:

1. Most texts (especially real-world sequences/texts) contain repeated substrings.
2. The order of the q-grams within equal substrings is the same.

The q-gram index therefore contains redundancy, which they aim to remove by extracting subsequences of the text.[24] The text is split into subsequences and occurrences of q-grams are only stored once for each distinct subsequence. This can save space because subsequences can in many cases be expected to occur repeatedly (especially in real world texts).
We implemented the q-gram/2L index in the class `IndexQGram2L`; our implementation is based on a prototype of a student's project by Merkle [Mer12]. Our implementation uses the basic idea of Kim et al. [Kim+05] but differs in some aspects mentioned below.

2.4.3.1 Data structure

Subsequences of fixed size h are extracted from the text at fixed intervals, where $h > q$ is a parameter of the index [Kim+05]. The subsequences are extracted in such a way that two neighboring sequences share an overlap of $q - 1$. (This overlap is necessary to not miss the q-grams at the borders of the subsequences.) The subsequences therefore start at text positions of the following form: $1 + i\,(h - q + 1)$ with $i \in \mathbb{N}$.[25] The set of all such subsequences is called *subsequences* and the number of distinct subsequences is denoted by $j := |\mathit{subsequences}|$ here.
The data structure consists of two main components (see also Figure 11):

- The *front-end index* stores for each q-gram in which subsequences it occurs (together with its offset in the subsequence).
- The *back-end index* stores for each extracted subsequence the positions in the text.

These two data structures make it possible to determine for a given q-gram all starting positions in the text (described in more detail in the following subsection). The subsequences themselves do not need to be stored, a unique identifier $\in [1, j]$ for each subsequence suffices.

We implemented the q-gram/2L index as follows:

- To implement the front-end index we can simply take the existing implementation of the classical q-gram and build it over the set *subsequences*. This index consists of a *directory* and a *positions* table (as described in the previous Section 2.4.1). In the *positions* table, the identifier of the subsequence is stored together with the offset of the q-gram in the subsequence.
- The back-end index could be implemented as h-sample index on the text with a step size of $h - q + 1$. However, this implementation would result in a *directory* of σ^h possible entries which could easily dominate the overall space consumption of the index in practice because $h > q$. Instead, we use an array of size j as *directory*, whose elements can be accessed with the identifiers of the subsequences.

(In the original proposal by Kim et al. [Kim+05] the index is stored among others using B^+-trees, but we chose to use the already existing efficient implementation of the q-gram index.)

[23] To harmonize naming we call it *q-gram/2L* here.
[24] The term *subsequence* is used here and in the original proposal to denote those extracted *contiguous substrings*.
[25] To simplify the exposition, we assume that the last subsequence is also of length h. If it is shorter, the text can be padded with additional characters.

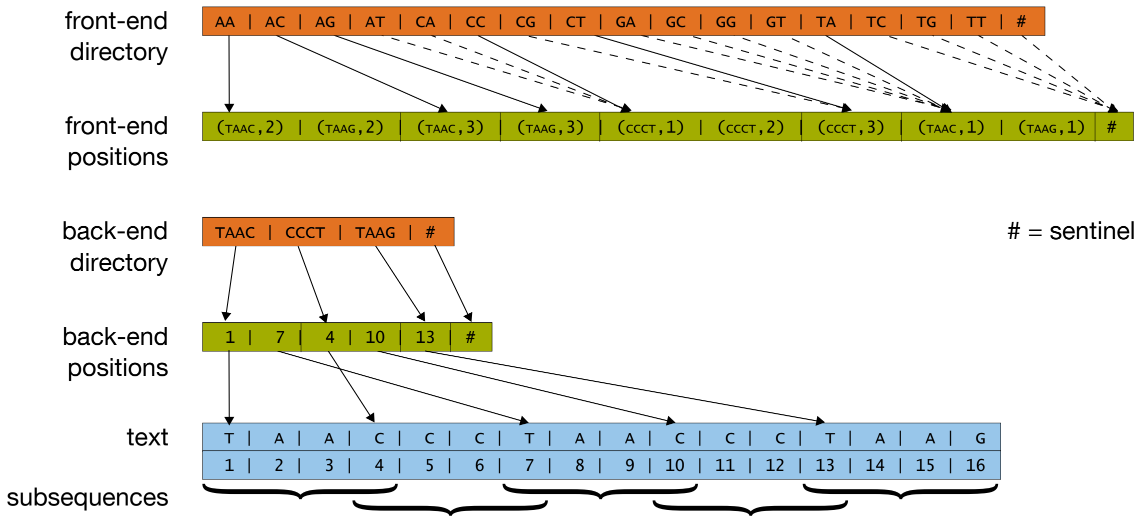

front-end directory
AA | AC | AG | AT | CA | CC | CG | CT | GA | GC | GG | GT | TA | TC | TG | TT | #
front-end positions
(TAAC,2) | (TAAG,2) | (TAAC,3) | (TAAG,3) | (CCCT,1) | (CCCT,2) | (CCCT,3) | (TAAC,1) | (TAAG,1) | #
back-end directory
TAAC | CCCT | TAAG | #
# = sentinel
back-end positions
1 | 7 | 4 | 10 | 13 | #
text
T | A | A | C | C | C | T | A | A | C | C | C | T | A | A | G
1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16
subsequences

Figure 11: *q*-gram/2L index: data structure.

The *q*-gram/2L index consists of a two-level structure. The front-end index maps each *q*-gram to all containing subsequences (stored as subsequence identifier and starting offset within the subsequence). The back-end index stores for each subsequence identifier the starting positions in the text.
In the example we have the gram size $q = 2$ and subsequence length $h = 4$, resulting in an overlap of 1 and *subsequences* = { "TAAC", "CCCT", "TAAG" } with $j = 3$.

The space consumption of the *q*-gram/2L index depends on the parameter h and on the number j of different subsequences. The front-end index has a *directory* with σ^q entries and the *positions* table stores values for j subsequences, each containing $(h - q + 1)$ *q*-grams. Each value consists of the subsequence identifier and the offset. The back-end index has a *directory* with j entries and the *positions* table stores all the $\left\lceil \frac{n}{h-q+1} \right\rceil$ sampled starting positions. All values are 4 B integers by default. This gives the following space usage of the *q*-gram/2L index (omitting constant summands):

$$S^{\text{front}}_{q\text{-gram/2L}} = S^{\text{front directory}}_{q\text{-gram/2L}} + S^{\text{front positions}}_{q\text{-gram/2L}} = \Big(\sigma^q + (h - q + 1) \cdot j \cdot 2\Big) \cdot 4\,\text{B}$$

$$S^{\text{back}}_{q\text{-gram/2L}} = S^{\text{back directory}}_{q\text{-gram/2L}} + S^{\text{back positions}}_{q\text{-gram/2L}} = \left(j + \left\lceil \frac{n}{h - q + 1} \right\rceil\right) \cdot 4\,\text{B}$$

$$S_{q\text{-gram/2L}} = S^{\text{front}}_{q\text{-gram/2L}} + S^{\text{back}}_{q\text{-gram/2L}}$$

We now want to provide upper and lower bounds for the size of the *q*-gram/2L which are independent of the number j of distinct subsequences. We do so by analyzing the worst case and the best case (assuming for simplicity that n is evenly dividable by $h - q + 1$ and that $\sqrt{n} = x \in \mathbb{N}$). In the *worst case*, all subsequences extracted from the text are different and no redundancy can be exploited. (A necessary condition is a sufficient alphabet size, otherwise sequences occur repeatedly.) The number of subsequences is $j = \left\lceil \frac{n}{h-q+1} \right\rceil$, yielding:

$$S^{\text{front}}_{q\text{-gram/2L}} = \Big(\sigma^q + 2n\Big) \cdot 4\,\text{B}$$

$$S^{\text{back}}_{q\text{-gram/2L}} = \left(2 \left\lceil \frac{n}{h - q + 1} \right\rceil\right) \cdot 4\,\text{B}$$

$$S_{q\text{-gram/2L}} = \left(\sigma^q + 2n + 2n \cdot \frac{1}{h - q + 1}\right) \cdot 4\,\text{B}$$

The *q*-gram/2L index therefore needs in the worst case several times the space of a classical *q*-gram index (Section 2.4.1).

In the (quite artificial) best case, all characters of the text are equal, e. g., t = "AAA. . .A". All extracted subsequences of length h are therefore equal as well, yielding $j = 1$. If we now choose the parameter h such that $h - q + 1 = \sqrt{n} \Leftrightarrow h = \sqrt{n} + q - 1$, the overall space usage is as follows:

$$S^{\text{front}}_{q\text{-gram/2L}} = \left(\sigma^q + (h - q + 1) \cdot 2\right) \cdot 4\,\text{B} = \left(\sigma^q + 2\sqrt{n}\right) \cdot 4\,\text{B}$$
$$S^{\text{back}}_{q\text{-gram/2L}} = \left(1 + \left\lceil \frac{n}{h - q + 1} \right\rceil\right) \cdot 4\,\text{B} = \left(1 + \sqrt{n}\right) \cdot 4\,\text{B}$$
$$S_{q\text{-gram/2L}} = \left(\sigma^q + 1 + 3\sqrt{n}\right) \cdot 4\,\text{B}$$

The q-gram/2L index therefore needs in the best case significantly less space than the classical q-gram index with $S_{q\text{-gram}} = (\sigma^q + n) \cdot 4\,\text{B}$.

Index parameters of `IndexQGram2L`:

1. `Q`: The length of the q-grams.
2. `TSpec`: `OpenAddressing` or `void` (direct addressing, default)
3. `subsequenceLength`: The length h of the extracted subsequences.

2.4.3.2 Construction

To build the two level structure, first the back-end index and then the front-end index is created:

- The back-end index is built by first creating the h-sample index with step size $h - q + 1$ using open addressing. Then the string set *subsequences* is created by traversing the entries. Simultaneously the *directory* is compacted so that it contains exactly one entry per subsequence.
- The front-end index is built afterwards by creating the q-gram index for the string set *subsequences*. Afterwards, *subsequences* is not needed anymore and can be discarded to save space.

The construction of the q-gram/2L index takes $T^{\text{construct}}_{q\text{-gram/2L}} = \mathcal{O}(n)$ time in the worst case because the construction of the front-end index and of the back-end index both need $\mathcal{O}(n)$ time (assuming q and h are constant).

2.4.3.3 Search

First we describe how to find the occurrences of a pattern of length $m = q$ in a text using the q-gram/2L index:

- The front-end index is used to determine the subsequences in which the q-gram occurs. First we compute the hash value of the q-gram and follow the corresponding pointer in the *directory*. The *positions* table contains the identifiers of the containing subsequences, together with the offset of the q-gram within each subsequence.
- The back-end index allows us to determine the text positions for each encountered subsequence identifier. By adding the offset to each such position, we can compute all occurrences of the q-gram in the text.

This algorithm gives the same results as when using the classical q-gram index to find the occurrences. However, the order of the positions might be different because here they are not necessarily sorted by increasing text position. Sorting can to be done afterwards if required.

For finding patterns shorter than q or longer than q the algorithm is extended just as with the classical q-gram index. For longer patterns, however, we modified the algorithm by implementing a heuristic to reduce the number of necessary verifications: When iterating over the containing subsequences for a q-gram we first check that the overlapping regions of the pattern and the subsequence are actually equal. If not, we can skip all occurrences of the q-gram for the current subsequence and immediately continue with the next subsequence.
The asymptotic search times are the same as for the classical q-gram index. Performing exact pattern matching of a pattern p with limited length $m \leq q$ in a text using a q-gram/2L index needs $T^{\text{bool/count}}_{q\text{-gram/2L}} = \mathcal{O}(q)$ and $T^{\text{pos}}_{q\text{-gram/2L}} = \mathcal{O}(q + occ)$ time in the worst case, independent of the size of the underlying text (which is optimal if $m = q$ or q is assumed to be constant).
Performing exact pattern matching of a pattern p with length $m > q$ in a text using a q-gram/2L index needs $T^{\text{bool/count/pos}}_{q\text{-gram/2L}} = \mathcal{O}(q + cand \cdot m) = \mathcal{O}(q + n\,m)$ time in the worst case, where *cand* denotes the number of matching candidate positions.

Extended search algorithms also for approximate pattern matching are described by Kim et al. [Kim+05] and Kim et al. [Kim+10].

2.4.3.4 Discussion

The two-level structure exploits regularities in the text to save space compared to the classical q-gram index. However, the subsequences are extracted at fixed intervals and are all of the same length. This is quite restrictive and prevents taking advantage of those repetitions that are not by coincidence aligned to the interval borders of the extracted subsequences.
The same authors therefore propose in Kim et al. [Kim+08] an extension of the q-gram/2L index, where subsequence of variable length are permitted (so-called *v-sequences*). The idea is to use the words of a natural language text as subsequences (short words are concatenated into longer strings, long words are split into smaller strings to achieve similarly sized subsequences). The subsequences are therefore allowed to start not only at fixed intervals. This method can save more space for natural language texts because words in general occur repeatedly. However, this approach works only if the text is actually structured in words (e. g., using whitespace and punctuation marks as delimiters). This is, for example, not the case for many types of biological sequences, such as DNA or proteins.
However, we believe it could be possible to use the same idea also for biological sequences: for DNA sequences we can, e. g., choose one of the characters $\{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ as word delimiter. The character with the most uniform distribution along the text should be chosen (so that there are no long stretches of the text where the character does *not* occur).
The q-gram/2L index was later modified and extended by Kim et al. [Kim+07] to use it for *approximate* pattern matching; the resulting index is smaller than the classical q-gram index and reported to be faster especially for high values of the search tolerance.

2.4.4 Other *q*-gram-based indexes

There are many different variants of q-gram based indexes, several of which try to reduce the space consumption. In the following we briefly sketch some interesting ideas (that have been or can easily be extended also for *approximate* pattern matching).
Navarro and Baeza-Yates [NBY98] propose a q-gram index targeted at natural language texts and approximate pattern matching. Instead of storing pointers to text positions it is possible to use *block pointers* which can significantly reduce the index size at the cost of a slower query processing.

Puglisi et al. [Pug+06] propose a compressed version of the q-gram index that also uses block pointers and differential encoding of the pointers. The resulting index is compared to a compressed version of the suffix array (namely that of Sadakane [Sad02]). Experiments show that the compressed q-gram index is faster at reporting occurrences of a search pattern if the number of occurrences is high.

Li et al. [Li+07] propose the VGRAM[26] index that relaxes the requirement that all indexed q-grams need to have the same length and store *variable* sized grams instead. The idea is based on the observation that some q-grams occur very frequently in real world texts; on the one hand, the posting list of such a frequent q-gram needs much space and, on the other hand, is not very useful for pattern matching because too many positions need to be verified. The goal is to choose a subset of all possible q-grams with lengths between q_{min} and q_{max} and include only this subset in the indexing (so-called *high quality grams*). Yang et al. [Yan+08] propose a variant of the VGRAM index optimized for approximate pattern matching.

Another independent variant of q-gram indexes with grams of variable lengths is proposed by Navarro and Salmela [NS09]. The goal is to obtain an index with position lists of roughly equal length. The index structure is constructed with the help of a suffix tree. The authors also describe algorithms for exact and approximate pattern matching using this index.

Behm et al. [Beh+09] propose another variant of q-gram indexes to reduce the space consumption by using two ideas: One idea is to merge the position lists of q-grams if they are sufficiently similar. The other idea is to discard lists of certain q-grams to save space. Both methods come at the cost of a more costly query processing but lead to a smaller index.

Transier and Sanders [TS08; TS10] propose a practical q-gram index for natural language texts using various compression techniques.

2.5 Summary

The index structures available in the implementation are summarized in Table 2. The performance of individual index structures in practice is evaluated in Section 6.3. Results for approximate pattern matching algorithms using the index structures are discussed in Section 6.5.

[26]VGRAM = **V**ariable-length **gram**s

Name	Index class	Reference	TUM[27]	Implementation	Section	Provides suffix tree iterator
SA	`IndexEsa`	Abouelhoda et al. [Abo+04]	○	Weese [Wee06]	2.1.1	●
SA + lcp	`IndexEsa`	Abouelhoda et al. [Abo+04]	○	Weese [Wee06]	2.1.2	●
WOTD	`IndexWotd`	Giegerich et al. [Gie+03]	○	Weese [Wee12]	2.2.4	●
STTD64	`IndexSttd64`	Halachev et al. [Hal+07]	●	Aumann [Aum11]	2.2.5	◐[28]
ESA	`IndexEsa`	Abouelhoda et al. [Abo+04]	○	Weese [Wee06]	2.2.6	●
DiGeST	`IndexDigest`	Barsky et al. [Bar+08]	●	Dau and Krugel [this work]	2.2.7	○
FMI	`FMIndex`	Ferragina et al. [Fer+04]	○	Singer [Sin12]	2.3.1	◐[29]
CSA	`IndexSadakane`	Sadakane and Shibuya [SS01]	●	Stadler [Sta11]	2.3.2	○
LZI	`IndexLZ`	Navarro [Nav04]	●	Stadler [Sta11]	2.3.3	○
q-gram	`IndexQGram`	Döring et al. [Dör+08]	○	Weese [Wee12]	2.4.1	○
q-sample	`IndexQGram`	Döring et al. [Dör+08]	○	Weese [Wee12]	2.4.2	○
q-gram/2L	`IndexQGram2L`	Kim et al. [Kim+05]	●	Krugel [this work]	2.4.3	○

Table 2: Summary: implemented index structures for strings.

[27] Contribution as part of our research project at the Technische Universität München.
[28] Our implementation of STTD64 provides only top-down iteration at the moment.
[29] The FM index provides *prefix tree* iteration instead.

3 Algorithms for approximate search

Many algorithms to perform approximate pattern matching have been devised in the past decades. They have been developed in different research communities (bioinformatics, database systems etc.) and for different applications (see Section 1.1). The algorithms also use different techniques and have different strengths and weaknesses. Some are, for example, fast in practice while some are mainly interesting for the concepts used.
The previous chapter presented several index structures and the corresponding algorithms to perform *exact pattern matching*. This chapter briefly introduces similarity and distance measures for strings (Section 3.1), presents online algorithms for *approximate pattern matching* working directly on the text (Section 3.2), and most importantly discusses algorithms that make use of index structures (Section 3.3).

3.1 String measures

Approximate pattern matching is based on a measure for the similarity or the distance of the strings involved. In some applications it is more convenient to use the concept of string similarity, in others it is more practical to frame the problem using string distances. Formally, a similarity measure $\phi : \Sigma^* \times \Sigma^* \to \mathbb{R}$ and a distance measure $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}$ are functions that map pairs of strings to real numbers (sometimes restricted, e. g., to $\mathbb{R}_0^+$ or the interval $[0, 1]$). The input strings are in the following denoted by r and s and their lengths by a and b, respectively. Similarity and distance measures can usually easily be converted into one another, for example, by negating or by using the reciprocal. (The implementation in the software library uses *similarity measures*, called *scoring schemes*; distance measures can be simulated by using negative similarity values.)
The concrete definition of such a function depends on the context, and especially on the error model of the application. If two strings are considered similar, a similarity function should give a high value and a distance function should give a low value, respectively. In many applications there are errors that disturb the data very often (e. g., common spelling mistakes, frequent genetic mutations, etc.); these should therefore usually not have a big impact on the similarity or distance [Kuk92]. Rarely occurring errors produce strings that are typically regarded as not so similar and should therefore have a higher influence on the value of the measure. (An extensive analysis of errors for natural language texts has been carried out by Kukich [Kuk92].)
In the following, we describe some of the most popular distance and similarity measures for strings. They vary in several dimensions: Some are very easy to describe and implement, others are better fitted to model the underlying errors. Some are designed for natural language texts, others for biological sequences. Some are oriented for the comparison of short strings (such as last names [Bra05]), others are designed to compare long sequences (such as to align whole chromosomes). Some have linear time complexity, others can be computed, e. g., in quadratic time only.

3.1.1 Hamming distance

A simple distance measure for two strings r and s is the *Hamming distance* δ_{Hamming} that simply counts the number of character positions where the two strings differ [Ham50]. The Hamming distance between strings r and s can formally be defined as:

$$\delta_{\text{Hamming}}(r, s) = \begin{cases} \left|\{\, i \in [1, a] \mid r_{[i]} \neq s_{[i]} \,\}\right| & \text{for } a = b \\ \infty & \text{otherwise} \end{cases}$$

It can be computed by a simple simultaneous scan over both strings in $\mathcal{O}(\min\{\, a, b \,\})$ time.

This measure is especially suited if the application wants to model only those errors where single characters get replaced with other single characters (these errors are also called *mismatches* or *differences*). The Hamming distance can, for example, not very well model spelling errors where characters can also be inserted or omitted. This measure is, however, well suited to model single nucleotide polymorphism (SNP, pronounced *snip*), a variation very common in DNA sequences and responsible for a large portion of the genetic variation, e. g., of humans [Con10].
The definition of the Hamming distance for strings with different lengths can also be changed, such that the shorter string is padded with spaces first.
In the implementation in the software library, the Hamming distance is not directly available as scoring scheme, but can be simulated by a variant of the edit distance where insertions and deletions are given very high costs (see the following Section 3.1.2).

3.1.2 Edit distance

The arguably most popular distance measure for strings is the *edit distance*. It can be seen as an extension of the Hamming distance where in addition to replacing characters also insertions and deletions of characters are modeled. This is useful in many applications that contain data with errors such as spelling mistakes, OCR errors, genetic mutations, etc. [Nav01]. There are several variants, here we describe the simple edit distance, the weighted edit distance, and a variant using extended operations.

Simple edit distance. The simple edit distance $\delta_{\text{edit}}(r, s)$ (also called *Levenshtein distance*) of two strings r and s is defined as the minimum number of operations needed to transform r into s, where allowed operations are:

- Delete a single character
- Insert a single character[1]
- Replace a single character with another character (also called *substitution* or *mismatch*).

The method to *compute* the edit distance of two strings is based essentially on the following three observations:

1. If one of the strings is empty, the distance equals the length of the other string.
2. If the last characters of both strings are the same, both characters can be dropped without changing the value of the distance.
3. If the last characters of both strings are different, there is a minimal sequence of operations transforming r into s (and the operations in the sequence are ordered from left to right in r). The last operation of this sequence is a deletion, an insertion or a substitution.

These observations can be used to formulate a recurrence that relates the edit distance of *prefixes* of the two strings.

$$\begin{aligned}
&\delta_{\text{edit}}(r_{[..i]}, s_{[..j]}) = i && \text{if } j = 0\\
&\delta_{\text{edit}}(r_{[..i]}, s_{[..j]}) = j && \text{if } i = 0\\
&\delta_{\text{edit}}(r_{[..i]}, s_{[..j]}) = \delta_{\text{edit}}(r_{[..i-1]}, s_{[..j-1]}) && \text{if } r_{[i]} = s_{[j]}\\
&\delta_{\text{edit}}(r_{[..i]}, s_{[..j]}) = \min \begin{cases} \delta_{\text{edit}}(r_{[..i-1]}, s_{[..j]}) + 1 & \text{(deletion)}\\ \delta_{\text{edit}}(r_{[..i]}, s_{[..j-1]}) + 1 & \text{(insertion)}\\ \delta_{\text{edit}}(r_{[..i-1]}, s_{[..j-1]}) + 1 & \text{(substitution)} \end{cases} && \text{if } r_{[i]} \neq s_{[j]}
\end{aligned}$$

[1]The operations *insertion* and *deletion* taken together are sometimes summarized as *indel*, especially in bioinformatics [Con10].

The last two formulas can be combined as follows, where $1_{r_{[i]} \neq s_{[j]}}$ is an indicator function that takes value 1 if $r_{[i]} \neq s_{[j]}$ and 0 otherwise:

$$\delta_{\text{edit}}(r_{[..\,i]}, s_{[..\,j]}) = \min \begin{cases} \delta_{\text{edit}}(r_{[..\,i-1]}, s_{[..\,j]}) + 1 & \text{(deletion)} \\ \delta_{\text{edit}}(r_{[..\,i]}, s_{[..\,j-1]}) + 1 & \text{(insertion)} \\ \delta_{\text{edit}}(r_{[..\,i-1]}, s_{[..\,j-1]}) + 1_{r_{[i]} \neq s_{[j]}} & \text{(substitution)} \end{cases}$$

Using this recurrence, the edit distance can be calculated with a dynamic programming algorithm and a matrix M of dimension $(a+1) \times (b+1)$ storing the values for the string prefixes [Gus97]. The result, i. e., the edit distance of r and s, can be found in the bottom right cell $M[a+1][b+1]$. The algorithm has a running time of $\mathcal{O}(a \cdot b)$ and needs $\mathcal{O}(\min\{a, b\})$ space because only one column or row needs to be stored simultaneously.
The algorithms is the basis of and closely related to the dynamic programming algorithm for online approximate pattern matching (Section 3.2.1).

In the implementation of the software library, the simple edit distance is available under the name `EditDistance` (synonymic `LevenshteinDistance`).

Weighted edit distance. The weighted edit distance $\delta_{\text{edit, weighted}}$ is a canonical extension of the simple edit distance. Instead of uniformly imposing costs of 1 for all operations, different costs can be assigned, either depending on the type of operation or also depending on the characters involved. This allows to model errors much more accurately and to distinguish between frequent and unusual errors. An exemplary instance of such a weighted edit distance is the *typewriter distance* where the costs for substituting one character with another depends on the distance of the keys on the keyboard (e. g., measured with Manhattan distance) [Kuk92].
To compute the weighted edit distance, the above recurrence has to be adapted as follows (and the base conditions of the recurrence have to be adapted similarly):

$$\delta_{\text{edit, weighted}}(r_{[..\,i]}, s_{[..\,j]}) = \min \begin{cases} \delta_{\text{edit, weighted}}(r_{[..\,i-1]}, s_{[..\,j]}) + c_{\text{delete}}(r_{[i]}) & \text{(deletion)} \\ \delta_{\text{edit, weighted}}(r_{[..\,i]}, s_{[..\,j-1]}) + c_{\text{insert}}(s_{[j]}) & \text{(insertion)} \\ \delta_{\text{edit, weighted}}(r_{[..\,i-1]}, s_{[..\,j-1]}) + c_{\text{substitute}}(r_{[i]}, s_{[j]}) & \text{(substitution)} \end{cases}$$

The functions $c_{\text{delete}}, c_{\text{insert}} : \Sigma \to \mathbb{R}_0^+$, and $c_{\text{substitute}} : \Sigma \times \Sigma \to \mathbb{R}_0^+$ map characters or pairs of characters to their respective costs and can be stored as an array and as a matrix.
The weights of the operations can, for example, also be learned from a training set (e. g., applied to OCR errors by Krugel [Kru08]).
The algorithm to compute the distance and its running time stay basically the same (except that some optimizations that can be made for the simple edit distance cannot be used for computing the weighted edit distance).

In the software library, the scoring scheme `Simple` allows to give different costs to the three types of operations (substitutions, insertions, and deletions) by assigning non-positive values. The scoring scheme `ScoreMatrix` can additionally use costs depending on the characters involved by employing a scoring matrix with non-positive scores. Both scoring schemes also provide additional functions for gaps in alignments which are described in more detail in the following subsection.

Extended operations. The edit distance does not have to be restricted to the above mentioned three operations for single characters. It can, on the contrary, be extended to include other more involved and problem-specific operations.

Typing errors frequently include transpositions of neighboring characters. The *Damerau-Levenshtein distance* $\delta_{\text{edit, swaps}}$, e. g., models such errors by adding a transposition operation (also called *swap*) with cost 1 to the simple edit distance [Nav01].
When using OCR, typical errors are merges or splits of consecutive characters (e. g., "rl" $\to$ "d" or "m" $\to$ "rn") [Kru08]. In such applications, the edit distance can be extended with the corresponding merge/split operations.
These extended operations can also be combined with weighted costs. The edit distance can be generalized even further to consist of a finite set of allowed operations with corresponding non-negative costs. The value of the distance is then the minimal cost of a sequence of operations (or ∞ otherwise) where the cost of a sequence is the sum of the individual operations. With this generalized definition it is, however, necessary to require that a substring is not changed more than once as noted by Navarro [Nav01]: Otherwise the computation of the edit distance would correspond to a rewriting system, which is not even computable in general.

A considerable extension are operations that allow to completely move around substrings. This is also called *interchange rearrangement* and several variants of this approach are possible, also using weighted operations as introduced by Kapah et al. [Kap+09].

3.1.3 Alignments and scoring matrices

Alignment scores ϕ_{align} are a way to better describe in particular the evolutionary similarity of biological sequences (e. g., DNA/RNA sequences or amino acid sequences of proteins). It is not a distance but a similarity measure giving higher scores to biologically related sequences having, for example, the same function in the organism or stemming from a common ancestor [NW70].
Matching identical characters in both sequences are given a positive score. Some single character substitutions in DNA or protein sequences can occur without significantly changing the biological function; these substitutions should therefore also be given a positive score (albeit possibly a slightly lower value than for matches). Other substitutions can, on the contrary, change or destroy the biological function of the sequence and should therefore be given a negative score.

The alignment score can be calculated with a recurrence very similar to the weighted edit distance (but the maximum of the alternatives has to be used because it is a measure for the similarity):

$$\phi_{\text{align}}(r_{[..\,i]}, s_{[..\,j]}) = \max \begin{cases} \phi_{\text{align}}(r_{[..\,i-1]}, s_{[..\,j]}) + c_{\text{delete}}(r_{[i]}) & \text{(deletion)} \\ \phi_{\text{align}}(r_{[..\,i]}, s_{[..\,j-1]}) + c_{\text{insert}}(s_{[j]}) & \text{(insertion)} \\ \phi_{\text{align}}(r_{[..\,i-1]}, s_{[..\,j-1]}) + c_{\text{substitute}}(r_{[i]}, s_{[j]}) & \text{(substitution)} \end{cases}$$

The functions $c_{\text{delete}}, c_{\text{insert}} : \Sigma \to \mathbb{R}$, and $c_{\text{substitute}} : \Sigma \times \Sigma \to \mathbb{R}$ map characters or pairs of characters to their respective scores and can be stored as an array and as a matrix (called *scoring matrix* or *substitution matrix* in bioinformatics), respectively. Pre-calculated scoring matrices are available for proteins where the score values were derived from statistical observations. These matrices exist in different versions and are called, e. g., *PAM (Point Accepted Mutation)* or *BLOSUM (Blocks Substitution Matrix)*; they are also available in the software library.
The classical algorithm to compute such an alignment of two sequences is the algorithm by Needleman and Wunsch [NW70] and very similar to the dynamic programming algorithm for the weighted edit distance. This algorithm has a running time of $\mathcal{O}(a\,b)$.
(Alignments are primarily used in biological contexts, but also in other areas, such as social sciences [AT00].)

Gaps. In evolutionary processes, longer stretches can be cut out of a sequence or inserted into a sequence without altering the biological function significantly (for example because a piece of

the DNA is spliced out and not translated into a protein). To also accommodate for such errors, the alignment score $\phi_{\text{align, gaps}}$ is extended with so-called *gap* operations where the insertion (or deletion) of many consecutive characters costs less than the insertion (or deletion) of each individual character. This can be achieved by assigning costs for starting a gap and for extending a gap (so called *affine gap costs*) [Gus97].
The recurrence and the algorithm have to be extended to also model such gaps, resulting in a running time of $\mathcal{O}(a\,b \cdot \max\{\,a, b\,\})$.

In the implementation, it is possible compute the alignment score of two strings by using the scoring schemes `Simple` and `ScoreMatrix`, and by assigning positive as well as negative values (in contrast to the edit distance where only non-positive values are allowed). Both classes additionally allow to define costs for opening a gap and extending a gap.

Other types of alignments. The described alignment score refers to so-called *global alignments* (*global* because both strings are compared in their entirety). There are also *local alignments* where substrings with high similarity are searched, and *semi-global alignments* where the start or the end of one of the strings has to be included. It is also possible to compute *multiple sequence alignments* of more than two sequences, especially to find regions which are conserved within several genetic sequences.
Popular algorithms to compute alignments in practical applications are based on heuristic methods (the most popular algorithm is probably the BLAST algorithm by Altschul et al. [Alt+90]). These heuristics might lead to non-optimal solutions but speed-up the calculation enormously.

3.1.4 *q*-gram based measures

The q-grams of a string can be used to build efficient index structures (Section 2.4) but they can also be used to define the similarity of two strings. This can be done by, e. g., counting the number of common q-grams but there are several other methods.
The similarity or distance of two strings r and s using q-grams is usually based on the sets of q-grams, $A := Q_q(r)$, $B := Q_q(s)$ (Q denotes the set of q-grams as defined in Section 2.4, page 56). The parameter q can also be set to $q = 1$ so that the 1-grams correspond to single characters. If favored, the strings can also be prefixed and appended with $q - 1$ special symbols \$ so that all characters occur in the same number of q-grams and the beginning and ending are accounted for as well (e. g., for $q = 3$ and $r =$ "neighbor" we use $r' =$ "\$\$neighbor\$\$" instead).

***q*-gram similarity.** There are several possibilities to define a *similarity measure* for two strings using q-grams, among others the *Jaccard index (Jaccard similarity coefficient)* ϕ_{Jaccard}, the *Overlap coefficient* ϕ_{overlap}, the *Sørensen-Dice coefficient* $\phi_{\text{Sørensen-Dice}}$, and the *Tversky index* ϕ_{Tversky} [Cha06; Li+08; Mad13]:

$$\begin{aligned}
\phi_{\text{Jaccard}}(r, s) &:= \frac{|A \cap B|}{|A \cup B|} \\
\phi_{\text{overlap}}(r, s) &:= \frac{|A \cap B|}{\min |A|\,, |B|} \\
\phi_{\text{Sørensen-Dice}}(r, s) &:= \frac{2\,|A \cap B|}{|A| + |B|} \\
\phi_{\text{Tversky}}(r, s) &:= \frac{|A \cap B|}{|A \cup B| + \alpha\,|A \setminus B| + \beta\,|B \setminus A|} \quad \text{for parameters } \alpha, \beta \in \mathbb{R}
\end{aligned}$$

(The cases where the denominator is 0 can be handled canonically.)
The measures yield similarity values between 0 and 1 and can also be converted to distance measures, e.g. as follows: $\delta_X := 1 - \phi_X$.

These similarity measure have the property that they count strings as similar, even if larger portions of the strings have been moved around. When comparing, e. g., the strings r = "approximation algorithm", s = "algorithmic approximation", they are considered rather similar (unlike when using, e. g., the edit distance), which can be desirable in some applications.

***q*-gram distance.** It is also possible to define a *distance measure* $\delta_{\text{q-gram}}$ using q-grams. This can be done by building for both strings r and s the so-called *q-gram profiles* G_r, G_s [Ukk92]. A q-gram profile is a vector of size σ^q that contains one entry for each possible q-gram over the alphabet. Each entry stores the number of corresponding q-grams contained in the string. The distance of the two strings can then be defined based on some kind of distance measure for the two vectors. Ukkonen [Ukk92] uses the L_1 distance (Manhattan distance):

$$\delta_{\text{q-gram}}(r, s) := \sum_{p \in \Sigma^q} |G_r[p] - G_s[p]|$$

The resulting distance measure for strings is a lower bound for the simple edit distance and can therefore be used as an approximation [Ukk92].

All described q-gram based string measures can be computed in $\mathcal{O}(a + b)$ worst case time (assuming σ and q are constant), which is substantially faster than many other measures.

3.1.5 Further measures

Jaro similarity is a measure developed for record linkage problems. Intuitively speaking, it counts the number x of matching characters in both strings (characters that have a counterpart in the other string "which is not too many positions away") and the number y of transpositions that appear between both strings. A formal definition is given by Winkler [Win90; Win06]. Based on those two values x and y, the similarity is computed using the following formula:

$$\phi_{\text{Jaro}}(r, s) := \begin{cases} 0 & \text{for } x = 0 \\ \frac{1}{3}\left(\frac{x}{a} + \frac{x}{b} + \frac{x-y}{x}\right) & \text{otherwise} \end{cases}$$

It has later been extended by Winkler [Win06] to give more weight to the characters at the beginning of the strings.

Phonetic measures describe the similarity or compute a code for strings, to compare them based on their phonetic properties. Examples are the *Soundex* and the *Metaphone* algorithm [Kuk92].

Regular expressions are no actual similarity or distance measures, but they can also be used for approximate string matching, e. g., by using wildcards or other extensions.

Algebraic comparisons can be performed if the characters can be treated as numbers (i. e., we can meaningfully perform operations such as subtractions on them). In this case, many more other kinds of measures are possible but not described here.

Other string measures are, e. g., based on *maximal matches* (the minimum number of characters that have to be removed from r, in such a way that the resulting string contains only substrings of s) [Ukk92], or the most frequent k characters [Sek+14].

String measure	Symbol	Running time	Metric	Adapt.	Implementation
Hamming distance	δ_{Hamming}	$\mathcal{O}(a+b)$	●	○	`Simple`[2]
Simple edit distance	$\delta_{\text{edit, simple}}$	$\mathcal{O}(a\,b)$	●	○	`EditDistance`
Weighted edit distance	$\delta_{\text{edit, weighted}}$	$\mathcal{O}(a\,b)$	◐	◐	`ScoreMatrix`[3]
Alignment score	ϕ_{align}	$\mathcal{O}(a\,b)$	◐	●	`ScoreMatrix`
Alignment score with gaps	$\phi_{\text{align, gaps}}$	$\mathcal{O}(a\,b\max\{a,b\})$	◐	●	`ScoreMatrix`
Jaccard index	ϕ_{Jaccard}	$\mathcal{O}(a+b)$	○	○	
Overlap coefficient	ϕ_{overlap}	$\mathcal{O}(a+b)$	○	○	
Sørensen-Dice coefficient	$\phi_{\text{Sorensen-Dice}}$	$\mathcal{O}(a+b)$	○	○	
Tversky index	ϕ_{Tversky}	$\mathcal{O}(a+b)$	○	○	
q-gram distance	$\delta_{\text{q-gram}}$	$\mathcal{O}(a+b)$	○	○	
Jaro similarity	ϕ_{Jaro}	$\mathcal{O}(a\,b)$	○	○	

Table 3: Similarity and distance measures for strings.

Running time states the time to compute the measure for two strings of length a and b.

Metric indicates whether a given measure is a metric (defined in Section 3.3.9; for *similarity measures*, the corresponding distance measure is considered): ● means it is a metric, ◐ it is a metric for certain parameters, ○ means it is not a metric.[4]

Adapt. describes the adaptability, i. e., how well the measure can be adjusted for a certain application by setting parameters. ● means high, ◐ medium, and ○ means no adaptability.

Implementation indicates the name of the implementation in the software library.

3.1.6 Discussion

The simple edit distance is a relatively straightforward measure that can easily be interpreted and applied to many different applications. The weighted edit distance can better model the frequency of errors, the edit distance with extended operation can better model certain types of other errors, and alignment scores can be defined even more problem specific. However, these three extensions come at the price of a more complex computation.
Another disadvantage of alignment scores and the *q*-gram based measures is that in general they are not a *metric*, which makes it impossible to apply certain optimizations in search algorithms (e. g., if the triangle inequality is not guaranteed). The properties of metrics are discussed in Section 3.3.9.
The measures based on *q*-grams are rather problem specific and in our view not as easy to interpret and intuitive as, e. g., the simple edit distance.

Even though a complex scoring model can possibly better describe application-specific string similarity, the simple edit distance is a very wide-spread and useful measure. It is arguably the most popular distance measure for strings and used in many of the solutions for approximate pattern matching. Sometimes the simple edit distance is also used as first approximation, and a more detailed computation with a problem specific similarity measure is carried out afterwards [Bra05]. In this thesis we therefore focus primarily on the simple edit distance.

The described similarity and distance measures are summarized in Table 3. Software libraries for computing string measures are listed in Section 4.4 and Table 5 on page 103.

[2]with high costs for insertion and deletion
[3]with only non-positive scores
[4]A subtlety is that some of the measures based on sets of *q*-grams constitute a metric when considering those *sets* as elements. But here we work on *strings*, and two different strings can yield the same sets of *q*-grams, contradicting the identity of indiscernibles.

3.2 Online approximate search

Algorithms for online approximate pattern matching do not need an index structure but operate directly on the text. No costly preprocessing of the text is needed and no big data structures have to be built beforehand (online algorithms therefore usually preprocess the *search pattern* in some way to carry out the search more efficiently). Even though this thesis concentrates on approximate pattern matching *using index structures*, we first discuss some *online* algorithms that are relevant in practice or interesting for their concepts. Navarro [Nav01] states that "*virtually all the indexed algorithms are strongly based on online algorithms*". Online algorithms are, for example, used as components of algorithms for indexes to verify potential matches.

The classical way for online approximate pattern matching is using a dynamic programming approach (Section 3.2.1), which can be sped-up by bit-parallelism (algorithm of Myers, Section 3.2.2). Entirely different approaches are splitting the pattern into pieces (PEX, Section 3.2.3) or building an automaton recognizing matching substrings (ABNDM, Section 3.2.4).
The algorithms use very different approaches and we outline the underlying idea, give some historic notes and argue why it is of interest here. For each algorithm we furthermore describe several aspects that are outlined in the following paragraphs.

String measure. The algorithms differ regarding the string measure that can be used. Some algorithms are restricted to a simple distance measure (and are therefore possibly faster), while other algorithms are flexible and can use several and application-fitted distance measures.

Search algorithm. The online search algorithms usually preprocess the pattern to build a data structure (which is relatively small compared to the text). The space consumption of an online algorithm X therefore normally depends on the pattern length m and is denoted by S_X. The worst case time to answer boolean, counting and position queries is usually the same and denoted by T_X. Online algorithms scan the whole text in some way from left to right and are therefore slower as the text gets longer, usually with a linear dependence on the text length.
Most online algorithms return the matching positions in the order from left to right. Some algorithm compute the end positions of the matching occurrences; if the starting positions are required they have to be computed in an extra step afterwards.
Some algorithms are parameterized which is described in the corresponding sections.

Implementation. The described online algorithms had already been implemented. In the implementation, approximate pattern matching works basically the same as exact pattern matching using subclasses of the `Finder` and `Pattern` class. The two classes store information about the text and the search pattern, respectively. For approximate pattern matching, the `Pattern` class has to be specialized and offers additional functions to specify the search tolerance /score limit (`setScoreLimit`) and the string measure (`setScoringScheme`) if different measures are supported by the algorithm. The type of the scoring scheme in some cases also has to be defined using a template parameter `TScore`. Finding the approximate matches then works the same as for exact pattern matching by calling the `find` function repeatedly until it returns `false`, and the text positions can be retrieved using the function `position()`.

Discussion. We address for each algorithm the advantages and disadvantages, describe variants of the algorithm, and discuss settings for which the algorithm performs best in practice (depending on the pattern length, alphabet size, etc.).

3.2.1 Dynamic Programming

Dynamic programming (DP) is the classical approach to *compute* the similarity or distance of two strings (see Section 3.1.2, Section 3.1.3 and also Gusfield [Gus97, Section 11]). This approach can be extended to *search* in a long text all substrings whose distance to the pattern is below a given threshold with the algorithm proposed by Sellers [Sel80]. It is closely related to the algorithm of Needleman and Wunsch [NW70] for computing a so-called *global alignment*, and the algorithm of Smith and Waterman [SW81] to compute a so-called *local alignment*. The DP algorithm is furthermore also the conceptual basis of several other algorithms for approximate pattern matching (online and offline) and therefore of particular importance.

The DP algorithm was contained in the software library and is available under the name `DPSearch`.

3.2.1.1 String Measure

The DP algorithm is very flexible regarding the distance/similarity function. In the implementation it can use the simple edit distance, a weighted edit distance, or a complex user defined scoring scheme, also with flexible gap costs. It is possible to use the Hamming distance together with the DP algorithm, but there are more specialized algorithms for this distance measure.[5]
The DP algorithm can also be extended to more complex variants of the edit distance by also modeling, for example, character swaps, other extended operations, or a more sophisticated scoring of gaps.

3.2.1.2 Search algorithm

The DP algorithm for approximate pattern matching is a simple extension of the algorithm to compute the edit distance (Section 3.1.2). We compute a matrix for the pattern and the text of lengths m and n; the matrix has dimension $(m+1) \times (n+1)$ and is computed column-wise by using the recursion of the edit distance. The only difference to the *computation* of the distance is that each cell in the first row of the matrix is initialized with 0, allowing the pattern to start at any text position. If a cell in the last row of the matrix contains a value of at most k, we have found the end position of an approximate match, because the distance of the pattern and a substring of the text ending at this position is at most k (Gusfield [Gus97, Section 11.6.5]).
If we are interested in the starting positions of matches we can search backwards from the found end position. In the implementation this is achieved by performing another DP computation.
Implemented as described above, the algorithm would take $\mathcal{O}(n\,m)$ space. However, during the computation of a column, only the previous column is needed. If not the whole matrix but only one column is stored, the algorithm has a space consumption of $S_{\text{DP}} = \mathcal{O}(m)$.
It takes $T_{\text{DP}} = \mathcal{O}(n\,m)$ time to compute the complete matrix. However, since we are only interested in parts of the matrix with values of at most k, there is an improvement by Ukkonen [Ukk85] (also called *Ukkonen cut-off*), leading to an expected running time of $T_{\text{DP with Ukkonen}} = \mathcal{O}(k\,n)$, maintaining a pointer to the so-called *last active cell*.

Algorithm parameters of `DPSearch`:

1. `TScore`: The scoring scheme.
2. `TFindBeginPatternSpec`: The algorithm to find the starting position of the match. Default: `DPSearch`

[5] In the implementation it is possible to simulate the Hamming distance by giving high costs to insertions and deletions (i. e., a negative score with high absolute value).

3.2.1.3 Discussion

The main advantage of the DP algorithm its flexibility regarding the string measure. The disadvantage is the rather high search time needed in theory as well as in practice. DP is therefore not competitive compared to other algorithms when using simple edit distance [Aïc06].
The DP algorithm can also be used to compute the traceback of the alignment (i. e., the sequence of operations necessary to transform the pattern into the match).
The DP algorithm constitutes the basis for many other algorithms such as the algorithm of Myers (next section) and an offline backtracking algorithm (Section 3.3.2). The algorithm can also be extended to search for multiple patterns simultaneously [Wee12].

3.2.2 Bit-parallel algorithm of Myers

The bit-parallel algorithm by Myers [Mye99b] is a variant of the DP algorithm using bit operations to speed-up the computations. It is also called, e. g., *algorithm of Myers*, *bit-parallel algorithm*, *bit vector algorithm by Myers*, and BPM [NR02]. The algorithm is interesting here because it runs fast in several practical applications.

An implementation was already available in the software library under the name `Myers`.

3.2.2.1 String Measure

The algorithm of Myers uses the simple edit distance as string measure, i. e., without weights or extended edit operations. Since the algorithm represents the matrix using bit vectors (see below), the similarity measure is rather restricted, each operation has to cost either 0 or 1.
To a limited extent it is possible to modify the algorithm for extended operations, which is done for example by Hyyrö [Hyy01] modeling transpositions of neighboring characters (character swaps) with a small slow-down in practice.

3.2.2.2 Search algorithm

The algorithm represents the DP matrix by encoding differences between neighboring cells in 5 bit vectors (of size m each). It is based on the observation that the difference of a cell to its top, left, and top-left neighboring cell can take only the value -1, 0, or 1 [Ukk85; Hyy01]. Additionally, the search pattern is represented by σ bit vectors (of size m each).
By using simple bit-operations (such as shift `<<`, bitwise and `&`, bitwise or `|`, and addition `+`), the matrix is computed column by column. If the pattern has at most the length of a machine word W, the bit-operations can be carried out very efficiently in parallel inside the CPU. The algorithm can be extended to also handle the case where the pattern is longer than a machine word [Mye99b; Hyy01]. The algorithm has a worst case running time of $T_{\text{Myers}} = \mathcal{O}(n\, m / W)$.
The implementation of this algorithm also makes use of the Ukkonen cut-off (see previous Section 3.2.1), extended here for blocks of the DP matrix. This optimization gives an expected running time of $\mathcal{O}(k\, n / W)$. The algorithm of Myers needs $S_{\text{Myers}} = \mathcal{O}(\sigma\, m / W)$ space.

Just as the DP algorithm, the algorithm of Myers finds the end positions of matches. It is possible to perform another computation to also find their starting positions.

Algorithm parameters of `Myers`:

1. `TFindBeginPatternSpec`: The algorithm to find the starting position of the match (default: `Myers`).

3.2.2.3 Discussion

The algorithm of Myers is fast, especially for short patterns with $m < W$, so that the bit operations can be used efficiently. The algorithm performs especially well compared to other online algorithms if the error level is high and the alphabet is small [Aïc06]. It is limited to the simple edit distance and cannot be used with weighted costs or other measures.

3.2.3 Splitting the pattern (PEX)

A completely different idea to find the approximate matches of a pattern in a text is to split the pattern into smaller pieces. If we want to find a pattern allowing k errors (using simple edit distance) and split the pattern into $j = k + 1$ pieces, each matching text occurrence must contain one of the pieces without changes (according to the pigeonhole principle). This can be extended to an arbitrary number j of pieces so that one of the pieces has to match with at most $\lfloor k / j \rfloor$ errors as noted in the book by Navarro and Raffinot [NR02, Section 6.5.1].
PEX is an algorithmic idea based on splitting the pattern and using above generalized pigeonhole principle [NR02].[6] There are several algorithms using this idea, among others by Wu and Manber [WM92b], Baeza-Yates and Perleberg [BYP92], Baeza-Yates and Navarro [BYN99], and Navarro and Baeza-Yates [NBY99].
The partitioning approach of PEX is interesting here because the algorithm is reported to perform well for certain problem classes (e. g., for big alphabets) [NR02]. The underlying idea can furthermore be extended to perform *offline* approximate pattern matching (Section 3.3.3).

The PEX algorithms was already contained in the software library and is available under the name Pex. The implementation was done by Aïche [Aïc06].

3.2.3.1 String Measure

The generalized pigeonhole principle holds for many distance measures, among others the Hamming distance and simple edit distance. It holds in particular if

- each error costs at least 1 and
- one error cannot change two pieces at a time.

If extended edit operations such as character transpositions are allowed with cost 1, this lemma does not necessarily hold any more because one error can affect two pieces. It furthermore does not hold if a similarity measure with positive and negative scores is used because a negative score in one piece could be compensated by a positive score in another piece. In the implementation, the distance measure is limited to the simple edit distance.

3.2.3.2 Search algorithm

The basic PEX algorithm works as follows: The pattern is split into $j \in \mathbb{N}$ pieces and an online algorithm for exact pattern matching of *multiple patterns* is used to find all occurrences of the pieces in the text. If a piece is found, the algorithm has to verify the region around the matching occurrence to make sure it is an actual match of the whole pattern. Because only portions of the text have to be verified, the algorithm belongs to the class of *filter algorithms* in the taxonomy of Navarro and Raffinot [NR02].
There are different variants of the PEX algorithm using this general theme; they differ in the number j of pieces, the choice of the multiple pattern algorithm, the algorithm used for verification, and the size of the region which is verified if a candidate match is found.

[6]The abbreviation *PEX* is not explained in [NR02], it probably stems from **p**artitioning and **ex**act search.

There are many possible choices for the multiple pattern algorithm, for example, the popular algorithm of Aho and Corasick [AC75]. Wu and Manber [WM92b] use the Shift-And algorithm by Baeza-Yates and Gonnet [BYG92], Baeza-Yates and Perleberg [BYP92] use the algorithm of Sunday [Sun90], and Navarro and Baeza-Yates [NBY99] use the Set-Horspool algorithm by [Hor80] (the latter two are variants of the Boyer-Moore algorithm for exact string matching). In the implementation, the algorithm of Wu and Manber [WM94] is used by default, but the algorithms Set-Horspool, Aho-Corasick and Shift-And are available as well.
Verifying the candidate positions can be done using the DP algorithm (Section 3.2.1). However, since the distance measure is limited to the simple edit distance in the implementation, the more efficient algorithm of Myers [Mye99b] (Section 3.2.2) is used by default.
Preparing the finder for the pieces needs $\mathcal{O}(m)$ time, e. g., when using the algorithm of Aho and Corasick [AC75]. Searching the pieces in the text needs $\mathcal{O}(n + cand)$ time where *cand* is the number of found candidate positions. Since there are $\mathcal{O}(n)$ candidate positions in the worst case, the resulting worst case running time of the PEX algorithm is $T_{\text{PEX}} = \mathcal{O}\big(m + n\,m^2 / W\big)$ when using the algorithm of Myers for verification.

Hierarchical verification. The basic PEX algorithm was extended by Navarro and Baeza-Yates [NBY99] using the *generalized* pigeonhole principle (see above). The goal is to spend even less time verifying potential matches because each verification needs quadratic time in the pattern length. Instead of splitting the pattern into $j = k + 1$ pieces, it is decomposed hierarchically. In the first step it is split into $j = 2$ pieces, so a match would contain one of the pieces with at most $k' = \lfloor \frac{k}{2} \rfloor$ errors. This splitting is continued recursively until the search tolerance is 0. The corresponding pieces are maintained in a tree structure where the root represents the whole pattern and the leaves are the pieces actually searched in the text. An algorithms for multiple patterns is used to find the pieces, just as in the basic algorithm. When a piece matches the text, not the whole pattern is verified. Instead, the hierarchical decomposition is followed upwards as long as it is successful. If the algorithm finally verifies the root node, a match was found.

Algorithm parameters of `Pex`:

1. `TVerification`: Determines whether the non-hierarchical or hierarchical variant is used (`NonHierarchical`, `Hierarchical`)
2. `TMultiFinder`: The algorithm to find the pieces in the text (default: `WuManber`).

3.2.3.3 Discussion

The PEX algorithm is limited to the simple edit distance in the implementation. It works better for larger alphabets and for longer patterns because the filtering criterion can be expected to be more effective [NR02; Aïc06]. The algorithm furthermore works better if the error level is low because for higher tolerances the pattern is split into many small pieces and the verification costs explode; in particular it should hold: $\alpha < 1/(3\log_\sigma m)$ [NR02]. In practical experiments the algorithms was found to work well for $\alpha < 0.3$ [NBY99; Aïc06]. The *hierarchical verification* is applicable also for higher error levels [NR02; NBY99].

When the pigeonhole principle is applied to *several patterns at once*, all pieces of the patterns can be searched simultaneously when scanning the text. This idea is used by Weese [Wee12] where the set of patterns is indexed using a *q*-gram index (note that not the text but the patterns are indexed). This algorithm is contained in the software library under the name `Pigeonhole`.

The ideas of PEX are also used for an *offline* partitioning algorithm (Section 3.3.3).

3.2.4 Backward automaton (ABNDM)

Another approach to online pattern matching is to use deterministic or non-deterministic finite automatons that recognize suffixes of the search pattern. This idea is used for exact searching in the algorithm *BNDM*[7] by Navarro and Raffinot [NR00]. The extension to approximate search is called *ABNDM*[8] and is described also by Navarro and Raffinot [NR00; NR02].
It was already implemented in the software library and is available under the name `AbndmAlgo`. The implementation was done by Aïche [Aïc06].

3.2.4.1 String Measure

The algorithm uses the simple edit distance to build an automaton recognizing pattern substrings with at most k errors. To a limited extent it is in principle possible to also model other edit operations such as character transpositions by modifying the underlying automaton.
If different weights for the edit operations are allowed, the space consumption of the algorithm grows drastically because every possible value of the distance is modeled by a set of states in the automaton. The ABNDM algorithm therefore usually uses only the simple edit distance and the implementation is also limited to this measure.

3.2.4.2 Search algorithm

The algorithm slides a window of size $m - k$ over the text (because a match consists of at least $m - k$ characters). Each window is scanned from right to left by using an automaton that recognizes any reverse prefix of p with at most k errors (therefore *Backward* DAWG). If the scanning reaches the beginning of the window, a candidate match was found. If a transition in the automaton is missing, the window is shifted forward (it can be shifted by several positions by maintaining an additional variable *last* storing the length of the longest matching prefix).
The non-deterministic automaton can be simulated by using bit vectors representing the set of active states and exploiting bit-parallelism when computing the transitions. This requires the pattern length m to be smaller or equal to the word size W. The algorithm can be extended to longer patterns (the implementations contains two versions, one for short and one for longer patterns).
The candidate positions have to be verified for a real match of the whole pattern by using another online algorithm. The original proposal uses an automaton, the implementation uses the algorithm of Myers [Mye99b] [Aïc06].
A detailed description of the algorithm can be found in the book of Navarro and Raffinot [NR02].

The algorithm maintains $k + 1$ bit vectors of size m each, yielding a space consumption of $S_{\text{ABNDM}} = \mathcal{O}(k\,m / W)$.
Sliding and scanning the window of size $\mathcal{O}(m)$ over the text of size n takes in total $T_{\text{ABNDM}} = \mathcal{O}\left(n\,m^2 k / W\right)$ time because for each character $\mathcal{O}(m\,k / W)$ time is needed in the worst case. An analysis of the *average case* complexity can be found in Navarro [Nav01].

3.2.4.3 Discussion

The ABNDM algorithm is limited to the simple edit distance and applicable only in limited scenarios. It works better for small patterns and best if $m < W$ to make use of the bit-operations [NR02]. It furthermore works better for small alphabets [NR02], low search tolerances, and is not usable for higher tolerances [Aïc06].

[7] BNDM = **B**ackward **N**ondeterministic **D**AWG **M**achine, DAWG = **D**irected **A**cyclic **W**ord **G**raph
[8] ABNDM = **A**pproximate **BNDM**

3.2.5 Further algorithms

There are several more algorithms for online approximate pattern matching and related problems. An overview of online algorithms for approximate pattern matching is given in the extensive survey of Navarro [Nav01] and in the book of Navarro and Raffinot [NR02]. Here we briefly describe some other and in our view interesting approaches.

The algorithm by Ukkonen [Ukk92] exploits the fact that the q-gram distance (Section 3.1.4) is a lower bound for the simple edit distance (Section 3.1.2) for finding potential matches. This result is improved by Hanada et al. [Han+14] with an average-case linear-time algorithm.

The software tool agrep[9] by Wu and Manber [WM92a] for the Unix operating system performs approximate pattern matching based on regular expressions. It contains implementations of many fast string matching algorithms and selects the algorithm based on the properties of the search query.

In some applications it is not necessary to definitely find all matches. In these cases it is possible to speed up the search by using heuristics that trade accuracy for performance. The probably most popular heuristic algorithm is the Blast[10] algorithm by Altschul et al. [Alt+90] used in bioinformatics. It is used to find approximate matches of a pattern in a sequence database. The algorithms proceeds by extracting substrings of the search pattern, linearly scanning the database, finding approximate matches of the substrings, and extending the possible matches. The algorithm cannot guarantee to find all matches, but performs very fast in practice (especially compared to computing the alignment using dynamic programming which is not practical for large DNA databases). Blast additionally offers several tools for searching different kinds of databases and applying and displaying statistical measures.

3.3 Approximate search in index structures

For *offline* approximate pattern matching problems we are allowed to preprocess the text before answering the search queries. We can therefore build an index structure (e. g., one of the indexes discussed in Chapter 2) to speed up the search queries. Several ideas for algorithms are not specific for one index structure but can by applied to several. They are therefore discussed here separately of the index structures.

A generic technique to perform approximate pattern matching is generating the so-called *neighborhood* of the search pattern regarding the distance measure (Section 3.3.1). Some algorithms are specific to a certain class of index structures, like tree-based indexes (Section 3.3.2). Algorithms based on splitting the search pattern into smaller pieces can be used with nearly all index structures (Section 3.3.3, Section 3.3.4, and Section 3.3.5). Other algorithms have been devised for suffix forests (Section 3.3.6) and for compressed indexes (Section 3.3.7).

In addition to those algorithms, there are also special *index structures* that were developed for approximate pattern matching (Section 3.3.8) and metric indexes (Section 3.3.9). Other index structures are designed for answering top-K-queries (Section 3.3.10).

A short survey of offline algorithms for approximate pattern matching is given by Navarro et al. [Nav+01] and an extensive survey for dictionaries was published by Boytsov [Boy11].

For each algorithm we discuss several aspects described in the following paragraphs.

String measure. Just as the online algorithms, some of the offline algorithms only support restricted distance or similarity measures while other algorithms are very flexible.

[9] agrep = **a**pproximate **grep**, grep = **g**lobally search a **r**egular **e**xpression and **p**rint

[10] Blast = **B**asic **l**ocal **a**lignment **s**earch **t**ool

Supported index structure. Because the algorithms work on top of an index structure they require the data structure to provide specific capabilities. Some algorithms only expect that the index supports exact pattern matching and use this abstract interface (this is possible with all indexes in the implementation); other algorithms need a tree-like traversal of the index structure and therefore only work with indexes that provide the corresponding functionality.

Search algorithm. When using an index, we do not need to scan the whole text from left to right. The matches therefore are not necessarily returned in this order but can be arbitrarily sorted. A natural lower bound to solve a position query of approximate pattern matching for a pattern of length m is $\Omega(m + occ)$. However, no "reasonably sized index"[11] with this search time is known [MN05a]. Navarro [Nav11] even believes that either the index size or the search time grows exponentially (in k, m, or σ), which to our knowledge has not been proven yet.

Implementation. The software library did not contain any algorithms to perform approximate pattern matching using index structures. We therefore implemented some of the most popular algorithms. They consist of specialized `Finder` and `Pattern` classes and can be accessed just as the online approximate and the offline exact search algorithms. (A short example program is in Section A.1).

Discussion. We discuss the advantages and disadvantages, as well as possible extensions and variants for each algorithm.

3.3.1 Neighborhood generation

A very generic approach to perform approximate string matching by using an index structure is *neighborhood generation*. The *k-neighborhood* of a string r regarding a distance measure δ and a tolerance k is the set of all strings that have a distance of at most k: $N_k(r) := \{\, s \in \Sigma^* \mid \delta(r, s) \leq k \,\}$. The definition is analogous for similarity measures.

An extreme case of a search algorithm is to simply generate all strings in the k-neighborhood of the pattern p and to look them up in the index. If one of the neighbors can be found in the index, the search pattern occurs in the text. The advantage of this approach is that it works with every index structure that supports exact search. The major drawback is, however, that the k-neighborhood of the search pattern contains very many elements ($\mathcal{O}\left(m^k \sigma^k\right)$ [Ukk93]). Because no reasonable performance can therefore be expected in practice, we did not implement this basic neighborhood generation algorithm.

To reduce the size of the neighborhood and to thereby reduce the time to lookup the neighbors in the index, the *condensed* (*super condensed*) neighborhood can be used. It only contains those strings of the k-neighborhood that do not have a proper prefix (proper substring, respectively) that is also in the k-neighborhood. Algorithms to generate the super condensed neighborhood are described by Russo and Oliveira [RO05; RO07].

In some algorithms, the pattern is first split into smaller pieces and the neighborhood of the individual pieces is used (instead of the neighborhood of the whole pattern). The matching positions of the pieces then have to be verified for a real match. This is, for example, used in the intermediate partitioning algorithm described below (Section 3.3.4).

There are many variants of neighborhood generation algorithms specialized for different index structures. Neighborhood generation using a trie (or suffix trie / suffix tree) is called *backtracking* and described in the following section. There are also neighborhood generation algorithms in

[11] with size $\in \mathcal{O}\left(n \log^h n\right)$ for some $h \in \mathcal{O}(1)$

conjunction with q-gram indexes, like e. g., the practical proposals by Hyyrö and Navarro [HN03] and Cao et al. [Cao+05].

3.3.2 Backtracking in tries and suffix trees

The online DP algorithm (Section 3.2.1 can be extended to perform approximate pattern matching in tree-shaped index structures (Section 2.2). The idea is to carry out a limited depth-first search (DFS) in the tree and to cut off subtrees where the given search tolerance k is exceeded. This pattern matching algorithm is called *backtracking* and works with tries for dictionaries as well as with suffix tree representations for texts (because a suffix tree can be seen as a trie of all text suffixes).
Approaches to perform approximate string matching based on this idea have been proposed by Jokinen and Ukkonen [JU91] for suffix automata[12], by Ukkonen [Ukk93] and Cobbs [Cob95] for suffix trees, and by Shang and Merrett [SM96] for tries and suffix tries [Nav+01]. In the taxonomy of Navarro et al. [Nav+01], this backtracking algorithm belongs to the neighborhood generation algorithms because the visited nodes of the trie correspond to the neighborhood of the pattern. We implemented the backtracking algorithm in the software library, it is available under the name `DPBacktracking`. The implementation was based on a prototype by Poppe [Pop10].

3.3.2.1 String Measure

The backtracking algorithm is very flexible regarding the distance measure because it is based on the DP algorithm. It can use the weighted edit distance, and extended operations can also be incorporated by modifying the algorithm. The Hamming distance can be simulated by giving very high costs to insertions and deletions.
One limitation compared to the DP algorithm is that only non-positive scores can be used in the scoring scheme; otherwise the condition of the cut-off (Case 2, see below) would not hold.

3.3.2.2 Supported index structures

The backtracking algorithm works on trie-like data structures. The index has to offer tree traversal function, e. g., "*Go from the current node to the child node labeled with character u.*" and "*Go up to the parent node.*". It therefore works in general with tries and suffix trees representations. In the implementation it works with the enhanced suffix array (`IndexEsa`, Section 2.2.6) and the WOTD tree (`IndexWotd`, Section 2.2.4). (The implementation of the STTD64 suffix tree provides at the moment only the functionality to go down up in the tree but not to traverse upwards.)
Using backtracking in a *suffix forest in external memory* (like the DiGeST index, Section 2.2.7) cannot be expected to be efficient because it needs a great number of I/O operations: already at the first level of the virtual suffix tree the algorithm needs to load σ many partial trees. We therefore did not implement the backtracking algorithm for suffix forests.

3.3.2.3 Search algorithm

Performing *exact* pattern matching in a dictionary using a trie is quite easy (Section 2.2.1): We simply have to process the pattern from left to right and descend the edge corresponding to the current character in each step. To perform *approximate* pattern matching, we additionally have to go down some "wrong branches" of the trie. A schematic example of the execution of the backtracking algorithm is shown in Figure 12.

[12]A *suffix automaton* (also known as DAWG) is the smallest deterministic finite automaton (DFA) recognizing all text suffixes. It can also be seen as a compacted form of the suffix tree where equal subtrees are merged. It is also used by the ABNDM algorithm in Section 3.2.4.

The goal is to find all nodes of the trie that have a path label r with $\delta(r,p) \leq k$. When searching a *dictionary* we are interested in matching leaves only. When searching a *text* using its suffix trie we are also interested in matching internal nodes because we are looking for matching infixes = prefixes of the stored suffixes. (To simplify the explanation, the algorithm is described primarily for a trie, but can easily be adapted, e. g., for a Patricia trie, a suffix trie, or a suffix tree.)
The algorithm starts at the root node and performs a limited depth-first traversal [Nav+01]. For each visited node the algorithm calculates the DP matrix of the pattern and the current path label. Assume the current node has the incoming edge label $u \in \Sigma_\$$ and the DP matrix of the parent node is available; then it is easy to compute the DP matrix of the current node by simply adding a column and using the recursive formula of the DP. We can distinguish three cases:

1. *Match:* The current node is a match if the bottom-right entry of the DP matrix is $\leq k$ because it represents the distance of the search pattern and the path label of the current node. If we are searching a trie and the current node is a leaf, we can output the current path label. If we are searching a suffix trie we have to output all positions stored in the leaves of the subtree rooted at the current node. This can be done by traversing the subtree or using a link to the first child if available (see Section 2.2.1).

2. *Cut-off:* No node below the current node will match the search pattern if all entries in the last column of the DP matrix are $> k$. We can then prune the whole subtree below the current node and continue with the next node in DFS order.

3. *Otherwise:* The algorithm continues with the next node in DFS order.

The algorithm maintains the columns (length $m + 1$) of the DP matrix in a stack. The stack grows and shrinks during the traversal in the trie. The following analysis is based on the simple edit distance as string measure. The maximal number of columns is $m + k$, the total space consumption of the algorithm is therefore $S_{\text{Backtracking}} = \mathcal{O}\left(m^2\right)$ in the worst case (assuming $k < m$ and the cost of a character insertion is at least 1).
The running time of the backtracking algorithm can be bounded using the size of the neighborhood of the pattern. The size of the neighborhood of a pattern with length m and edit distance k is analyzed by Ukkonen [Ukk93] to be $\leq \frac{12}{5}(m+1)^k(\sigma+1)^k = \mathcal{O}\left(m^k\sigma^k\right)$ (the alphabet size is here not assumed to be constant). There are $\mathcal{O}(m)$ nodes on the path to each neighbor. Each node has a processing time of $\mathcal{O}(m+\sigma)$ for calculating the DP column and pruning the children. This gives an overall running time of $T_{\text{Backtracking}} = \mathcal{O}\left(m^{k+2}\sigma^{k+1} + occ\right)$.
There are several papers analyzing the search time of approximate pattern matching in tries and related problems. The *expected* time to perform approximate pattern matching in a trie is reported to be $\mathcal{O}\left(k\sigma^k\right)$ by Shang and Merrett [SM96] when using edit distance. The average time time for Hamming distance (and related measures) is analyzed by Maaß [Maa04]. The worst case search time of suffix trees and string-B trees in external memory is analyzed by Mühling [Müh08]. Kristina Bayer [Bay12] analyzed exactly (not asymptotically) the number of visited nodes when performing approximate pattern matching in a suffix trie (for $k \leq 2$).

Algorithm parameters of `DPBacktracking`:

1. `TScore`: The scoring scheme.

3.3.2.4 Discussion

An advantage of the backtracking algorithm is its flexibility regarding the distance measure: the simple and weighted edit distance can be used and other edit operations can be incorporated by

modifying the algorithm.
A drawback of the backtracking algorithm is that an index structure has to be built beforehand, and that the index structure needs to support tree traversal operations (it is not sufficient to permit exact searching).
The running time of the backtracking algorithm is independent of the text length which makes it especially appealing for long texts. The search time only depends on the search tolerance k and the pattern length m (and on the alphabet size σ if not assumed to be constant). The search time can be expected to deteriorate for larger alphabets and for longer patterns because of the exponential dependence.
Unlike other algorithms, no parameter tuning is necessary for the algorithm to work.

There are several variants of the basic backtracking algorithm. Many attempt to cut off more subtrees to avoid processing irrelevant nodes that do not lead to a match. This can be done by spending more time in each node computing cut-off heuristics [Nav+01]. One algorithm of this kind is by Rheinländer et al. [Rhe+10], reporting an improved search time by pruning subtrees based on the pattern length and several other factors.

Another variant of the backtracking algorithm has recently been proposed by Siragusa et al. [Sir+13a; Sir+13b] to search *multiple patterns* at once. The algorithm is also included in the software library and builds one index on the text and additionally another index on the set of patterns. The backtracking is then performed simultaneously in both indexes for (virtual) suffix tree nodes.

3.3.3 Partitioning into exact search

Performing approximate pattern matching by splitting the pattern into smaller pieces is used in the online algorithm PEX (Section 3.2.3) but can also be used in an offline algorithm. Instead of finding the pieces with a multi-pattern online algorithm, the pieces can be searched in an index structure.
The search pattern is split into $j \geq k + 1$ pieces, so each approximate text occurrence has to contain at least one piece without errors. The pieces are therefore searched in the index and all found candidate positions are verified for a real match. This approach is usually called *partitioning into exact search* and is used by many different offline approximate pattern matching algorithms. An overview is given in the survey of Navarro et al. [Nav+01]. The partitioning approach is especially interesting because it works with every index structure that allows exact search.
We implemented this algorithm in the software library based on a prototype by Poppe [Pop10]. The algorithm is available under the name `Partitioning<IntoExactSearch>`.

3.3.3.1 Similarity measure

The algorithm works with several distance measures, in particular with Hamming and simple as well as weighted edit distance. The same restrictions as for PEX apply (Section 3.2.3).

3.3.3.2 Supported index structures

The partitioning algorithm works with all index structures (the index only needs to provide exact search functionality).

Note that if the index structures imposes a restriction on the length of the pattern, this applies here for the *pieces* of the pattern. The q-gram and q-gram/2L indexes with enabled open addressing require that the search pattern is of length at least q (see Section 2.4.1 and Section 2.4.3). When using the partitioning algorithm with $j = k + 1$ it therefore has to hold: $\lfloor m/j \rfloor \geq q \Leftrightarrow m \geq q(k + 1)$.

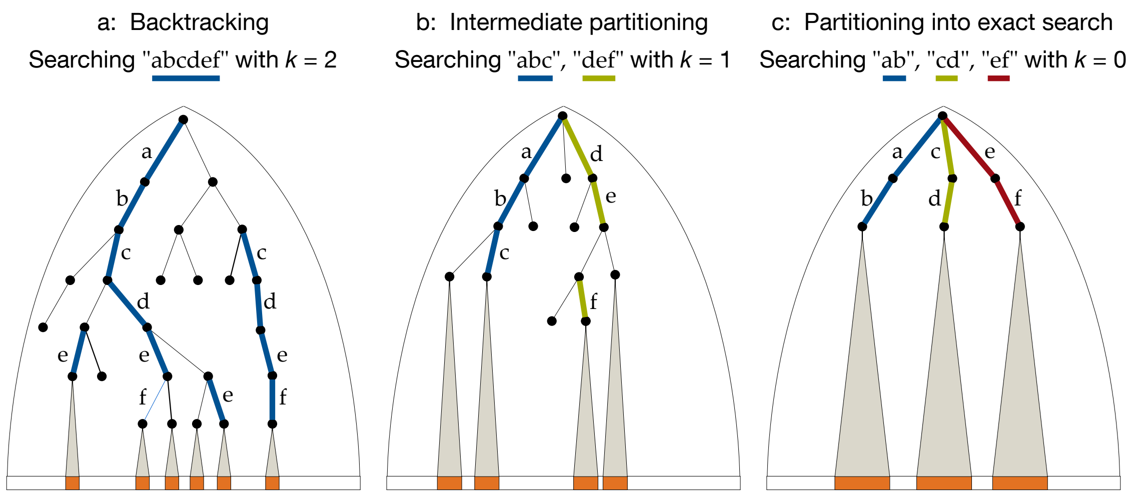

a: Backtracking
Searching "abcdef" with k = 2
b: Intermediate partitioning
Searching "abc", "def" with k = 1
c: Partitioning into exact search
Searching "ab", "cd", "ef" with k = 0

Figure 12: Approximate search: schematic example for the execution in a trie.

The pattern p = “abcdef” is searched in a suffix trie using the simple edit distance δ_{edit} and $k = 2$. Only the traversed part of the trie is shown. Matching edges are shown in bold with character labels; for non-matching edges no character labels are shown for readability. The leaves of the trie (in the case of a suffix trie corresponding, e. g., to the suffix array) are shown at the bottom. (The figure is partially based on [Nav+01].)

The *backtracking algorithm* (a) has to traverse the trie using the full search tolerance k so that many nodes have to be visited. The leaves in the subtrees of all found nodes are matches (they do *not* need to be verified).
The *partitioning into exact search algorithm* (c) splits the pattern p into $j = k + 1$ small pieces and each piece is searched without errors, visiting only a small number of trie nodes. Each found piece position is a candidate match of the whole pattern and the corresponding text area needs to be verified using an online algorithm.
The *intermediate partitioning algorithm* (b) splits the pattern into $2 \le j \le k$ pieces and each piece is searched using backtracking with a lower tolerance $k' = \lfloor k / j \rfloor$. The parameter j therefore allows a trade-off between the time to traverse the trie nodes and the time to verify the candidate matches.

The q-sample index only guarantees to find all patterns of length at least $q + stepSize - 1$. When using the partitioning algorithm with $j = k + 1$ it therefore has to hold: $\lfloor m / j \rfloor \ge q + stepSize - 1 \Leftrightarrow m \ge (q + stepSize - 1)\,(k + 1)$

3.3.3.3 Search algorithm

The basic partitioning algorithm works as follows. The search pattern is split into $j \ge k + 1$ pieces (in the implementation, the pieces are of preferably equal lengths $\lfloor m / j \rfloor$ and $\lceil m / j \rceil$). The index is used to find the starting positions of each piece in the text by performing an exact search. A schematic example of the execution of the partitioning algorithm is shown in Figure 12. For each found starting position of a piece, the corresponding text region is verified for an actual match of the whole pattern. The verification region is shown in Figure 13. The verification can be carried out by using any suitable online algorithm, e. g., the algorithm of Myers for simple edit distance or the DP algorithm for weighted edit distance (Section 3.2).

One technical detail regarding this algorithm is how to handle duplicates. If the positions of the matching pieces are guaranteed to be retrieved in order of increasing text position, they could be excluded by simultaneously traversing the position lists of the pieces. However, since the index structures do not guarantee to return the positions in any specific order (except for the q-gram index for pattern of length q), we decided to store the found positions of the pattern and filter out duplicates this way.

For the analysis we assume to use index structure X and verification algorithm Y. The space usage is dominated by the space of the verification algorithm and by the data structure to filter duplicates: $S_{\text{Partitioning (exact)}} = S_Y + \mathcal{O}(cand)$.

The search time of the partitioning algorithm consists of the time to find the pieces of length $\leq \lceil m/j \rceil$ each, plus the time to verify the *cand* found candidate regions of length $m+3k$: $T_{\text{Partitioning (exact)}} = \mathcal{O}\big(j \cdot T_X^{\text{pos}}(\lceil m/j \rceil) + cand \cdot T_Y^{\text{pos}}(m+3k)\big)$.
If the index supports exact search in $\mathcal{O}(m)$ worst case time and the DP algorithm is used for verification this yields, for example, $\mathcal{O}(m + cand \cdot m \cdot (m+3k)) = \mathcal{O}\big(cand \cdot m^2\big) = \mathcal{O}\big(n\,m^2\big)$.

Algorithm parameters of `Partitioning`:

1. `TSpec = IntoExactSearch/Intermediate/PartitioningHierarchical`: Determines which variant to use.
2. `TScore`: The scoring scheme.
3. `TPieceFinderSpec` and `TPiecePatternSpec`: The algorithm to search the pattern pieces in the index. (e. g., `Default/DPBacktracking/Partitioning`).
4. `TVerifyFinderSpec` and `TVerifyPatternSpec = Myers<FindInfix>`: The algorithm to verify a candidate position.
5. `setNumberOfPieces`: The number of pieces for splitting the pattern.
6. `DoPreparePatterns = False/True`: Switches between the standard partitioning (`False`) and the partitioning specialized for suffix forests (`True`, Section 3.3.6).

3.3.3.4 Discussion

The principal advantage of the partitioning algorithm for approximate pattern matching is its general applicability. It can be used with every index structure supporting exact pattern matching (all indexes in the implementation of the software library). It can furthermore be used with different string measures, among others, the simple and weighted edit distance.
The characteristics of this algorithm regarding the behavior in different settings can be expected to be similar to the PEX algorithm because they are both based on the same idea. It performs better for low error levels and long patterns because otherwise the pieces are too short and there are many candidate matches to verify [Nav+01]. The algorithms performs better for larger alphabets because the selectivity of the filter is better.

There are several variants of this basic partitioning algorithm. One way of optimizing the algorithm is to deliberately choose a splitting that promises only little verifications. This is done, for example, by Navarro and Baeza-Yates [NBY98] exploiting the characteristics of natural language texts.
In the basic variant, the pattern is split into exactly $k+1$ pieces [NR02]. It is also possible to split the pattern into *more* pieces and to merge the respective positions of the found pieces; a verification then only has to be performed if the number of matching pieces for a candidate region is high enough [Li+08]. This approach in general works better if the positions of the pieces are already sorted by text position because the position lists can be merged more efficiently.
Another possibility is to split the pattern into *fewer* pieces and to therefore allow a certain amount of error in each piece. Depending on the way to approximately find the pieces, this approach is called *hierarchical verification* (if the pieces are partitioned recursively, described in Section 3.3.5) or *intermediate partitioning* (if the pieces are searched using backtracking, described in Section 3.3.4).
We designed the implementation so that it is possible to modularly plug-in different algorithms for searching the pieces. This way we can easily adapt the basic algorithm to those two extensions.

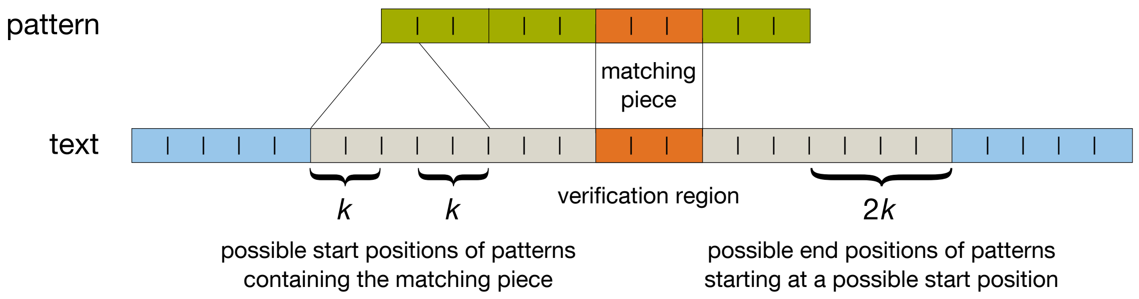

pattern
text
matching piece
k
k
verification region
$2k$
possible start positions of patterns containing the matching piece
possible end positions of patterns starting at a possible start position

Figure 13: Approximate search with partitioning into exact search.

The pattern is split into pieces and the positions of each piece in the text are retrieved using an index. For each matching piece, a region of the text has to be verified for a match with the whole pattern. The possible starting positions of the match are $\pm k$ positions around the projected starting position of the pattern because there can be at most k insertions and deletions.

If we want to be sure to find all end positions of patterns *starting at one of the possible starting positions* (not necessarily containing the matching piece), the verification region has to be extended up to $2k$ positions behind the projected end.

(In the implementation, the algorithm actually first searches for end positions and then finds the corresponding starting positions, so the picture is mirrored there.)

3.3.4 Intermediate partitioning

The basic pigeonhole principle guarantees that a matching occurrence has to contain one piece without errors if the pattern is split into $k + 1$ pieces. It is used for online search in the PEX algorithm, also in its hierarchical variant with a generalized pigeonhole principle. This generalized principle can also be used for *offline* searching and the pattern is split into fewer pieces ($j \leq k$) which are then searched in the index with a lower tolerance $k' < k$ using backtracking. This algorithm is called *intermediate partitioning* by Navarro and Baeza-Yates [NBY00] and also described in the survey of Navarro et al. [Nav+01].
We implemented this algorithm in the software library based on a prototype by Poppe [Pop10]. The algorithm is available under the name `Partitioning<Intermediate>`.

3.3.4.1 Similarity measure

The algorithm can be used with the same similarity measures as the PEX algorithm (Section 3.2.3) and partitioning into exact search (Section 3.3.3).

3.3.4.2 Supported index structures

The intermediate partitioning algorithm works with all index structures that support the backtracking algorithm (`DPBacktracking`, Section 3.3.2). This is at the moment true for the enhanced suffix array (`IndexEsa`, Section 2.2.6) and the WOTD tree (`IndexWotd`, Section 2.2.4).

3.3.4.3 Search algorithm

The number of pieces is a parameter of the intermediate partitioning algorithm. The pieces are of lengths $\lfloor m/j \rfloor$ and $\lceil m/j \rceil$ and have to be searched in the index using a tolerance of $k' = \lfloor k/j \rfloor$.[13] Otherwise the search algorithm works just as the partitioning into exact search algorithm. A schematic example of the execution of the intermediate partitioning algorithm in relation to backtracking and partitioning into exact search is shown in Figure 12.

[13]Due to the generalized pigeonhole principle we can round down here and this is actually one important advantage of the intermediate partitioning algorithm because it allows to reduce the search tolerance.

The space and search time of the intermediate partitioning algorithm can be analyzed just as for exact partitioning. The search consists of the time to find the pieces of length $\leq \lceil m/j \rceil$ each using tolerance $\lfloor k/j \rfloor$, plus the time to verify the *cand* found candidate regions of length $m+3k$: $T_{\text{Partitioning (intermediate)}} = \mathcal{O}\big(j \cdot T_X^{\text{pos}}(\lceil m/j \rceil) + cand \cdot T_Y^{\text{pos}}(m+3k)\big)$.
A more detailed analysis of the search time is given by Navarro and Baeza-Yates [NBY00].

The implementation is generalized so that all partitioning algorithms use the same code base and differ only where necessary. To choose an algorithm for approximate search in the index structure, the parameters `TPieceFinderSpec` and `TPiecePatternSpec` can be used.

3.3.4.4 Discussion

The number of pieces allows a trade-off between spending time searching the index (when there is only 1 piece in the extreme case) and spending time verifying possible occurrences (when the pattern is split into $> k$ pieces). The backtracking algorithm (for $j = 1$) and partitioning into exact search (for $j = k + 1$) can be seen as extreme cases of the intermediate partitioning algorithm as described in the survey by Navarro et al. [Nav+01]. For many inputs it therefore performs better than these two algorithms and works for a broader range of k and m. This comes at the cost of tuning the parameter j that defines the number of pieces to split the pattern.

3.3.5 Partitioning with hierarchical verification

Another variant of the basic partitioning algorithm is called *partitioning with hierarchical verification* and aims at reducing the time spent for verifying candidate positions. The basic partitioning algorithm compares the *whole* search pattern with the corresponding text area as soon as one matching piece is found; these verifications can be quite costly especially if the search pattern is long and the tolerance is high. The *hierarchical verification* algorithm therefore tries to verify only smaller regions of the pattern with a lower tolerance as soon as one piece matches. This is the same idea as used by the hierarchical variant of the online algorithm PEX (Section 3.2.3) and can also be used to search in index structures [Mye94; HN03; Rus+09b].
We implemented this algorithm in the software library and the algorithm is available under the name `Partitioning<PartitioningHierarchical>`.

3.3.5.1 Similarity measure

The algorithm can be used with the same similarity measures as the PEX algorithm (Section 3.2.3) and partitioning into exact search (Section 3.3.3).

3.3.5.2 Supported index structures

The hierarchical verification algorithm can be used with all index structures supporting exact search. If the index, however, imposes a restriction on the pattern length for exact search, this restriction has to hold for all *pieces of pieces* of the search pattern.

3.3.5.3 Search algorithm

Just as the intermediate partitioning algorithm, the pattern is split into fewer pieces ($j \leq k$) so that each occurrence of a piece has to be searched with a tolerance of $k' = \lfloor k/j \rfloor$. To approximately find the pieces, they are again split into yet smaller pieces and so on until the tolerance is 0 and the pieces can be searched using exact pattern matching. If a matching piece is found, not the whole pattern is directly verified for an actual match. Instead, only the next larger containing piece is verified; if the verification is successful it is continued for the next larger piece and so on until

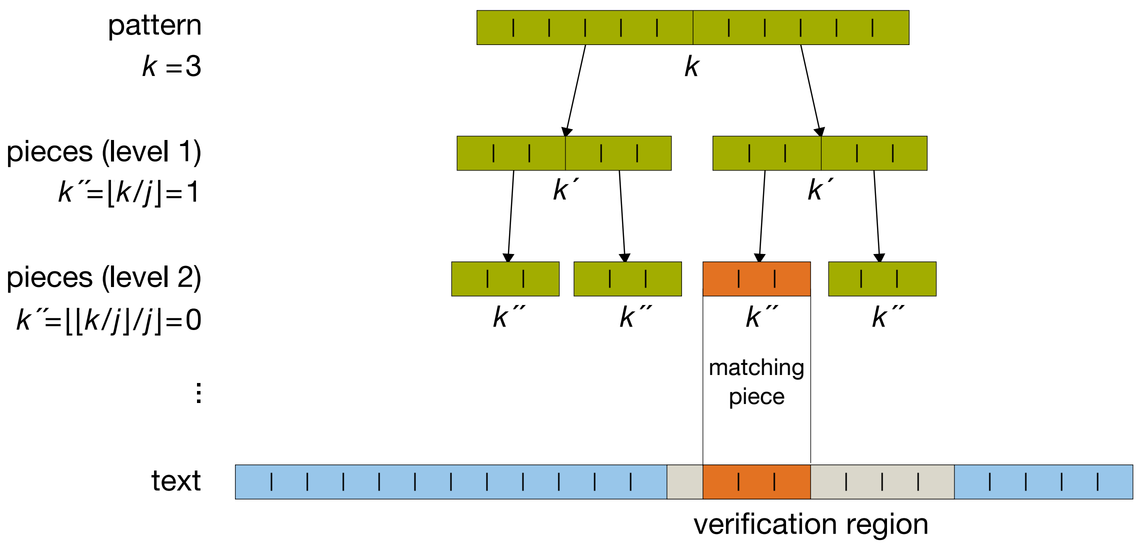


Figure 14: Approximate search with partitioning and hierarchical verification.

The occurrences of the pattern with tolerance k are searched. The pattern is therefore split into j pieces so that each piece has to match with tolerance $k' = \lfloor k/j \rfloor$. The pieces are recursively split again into pieces etc. until the search tolerance is 0 and the pieces can be searched using exact search. If a piece is found, the next larger piece of the hierarchical decomposition is used for verifying with the text using its tolerance. Only if it is a match, the verification tree is followed upwards until finally the whole pattern is verified.
This hierarchical splitting and verification process is shown here for $k = 3$ and $j = 2$, yielding two levels. (The figure is partially based on [HN03].)

eventually a verification fails or the whole pattern has been verified [HN03]. This process is shown in Figure 14.

For the analysis we assume that the number j of pieces is the same for all levels and that $k + 1 = j^h$ for some $h \in \mathbb{N}$. Then $h = \log_j (k+1)$ is the number of levels of the partitioning because we have to divide the pattern recursively until there are $k + 1$ pieces which can be searched without errors. We assume that the pattern length m is evenly dividable by $j, j+1, \ldots, j^h = k+1$ for the ease of presentation (this does not change the asymptotic behavior). The pieces which are searched exactly in the index are of length $\frac{m}{j^h}$. All these j^h exact searches together take $\mathcal{O}\left(j^h \cdot T_X^{\text{pos}}(\frac{m}{j^h})\right)$ worst case time.

For each found piece at the lowest level, the algorithm has to perform in the worst case (a true match) h verifications of differently sized regions. However, the verification time of the top-most level with a region of size $m + 3k$ dominates. This yields basically the same asymptotic time as partitioning into exact search: $T_{\text{Partitioning (hierarchical)}} = \mathcal{O}\left(j^h \cdot T_X^{\text{pos}}(\frac{m}{j^h}) + cand \cdot T_Y^{\text{pos}}(m + 3k)\right)$, except that the number of pieces j is replaced with j^h.

3.3.5.4 Discussion

Partitioning with hierarchical verification can be significantly faster than partitioning into exact search if there are many candidate positions and most of the verifications fail before the top-most level is reached.

The described hierarchical verification idea is used among others by the algorithm of [Mye94] together with a q-gram index, and based on that by Hyyrö and Navarro [HN03] for a practical index structure for approximate pattern matching. Russo et al. [Rus+09b] use hierarchical verification to search in compressed indexes.

3.3.6 Approximate search in suffix forests

Approximate pattern matching in suffix forests and external memory suffix trees in general is only little studied so far. We first discuss the applicability of the backtracking and partitioning algorithms and then propose a new algorithm based on partitioning into exact search for multiple patterns.

The book by Barsky et al. [Bar+11a] describes a *backtracking* algorithm (Section 3.3.2) and states that the whole suffix forest needs to be traversed to answer an approximate search query, which is very expensive if the partial trees reside on disk. We think this could be improved by refraining from loading partial suffix trees that cannot contain a match based on the information stored in the dividers. But for answering an approximate query with edit distance and search tolerance k, the algorithm still would need to traverse at least σ^k nodes to handle the case of k errors at the beginning of the pattern (assuming the virtual suffix tree is dense in the first k levels, which is reasonable since k is usually very small). If the partial suffix trees are sufficiently small, all those nodes reside in different trees and each node access leads to one I/O operation. (If the partial suffix trees are bigger, the algorithm might need less I/O operations, but loading one tree therefore takes longer.) The backtracking algorithm and the disk-based storage of the partial suffix trees are therefore not a very efficient combination.

The strategy of *partitioning* the pattern into exact search (Section 3.3.3) seems to be a more promising alternative in terms of necessary I/O operations: For each of the $k + 1$ pieces of the pattern, one partial suffix tree (or a small number of consecutively stored trees) needs to be loaded into main memory, the piece is searched by descending from the root and the found positions are verified. With the existing layout it is, however, probably not possible to reduce the number of I/O operations for one query much because the matches are distributed in several partial trees, inevitably leading to several I/O operations.

We therefore propose an algorithm that builds on top of the partitioning into exact search algorithm. It does not handle each pattern individually, but preprocesses a *set of search patterns* to improve the overall search time of all queries. The aim is that each I/O operation helps answering *several queries* instead of only one. In contrast to most other mentioned algorithms, this algorithm therefore solves a different problem (*multi-pattern approximate string matching*). The implementation therefore has different interface: before answering the actual queries, the set of all patterns needs to be preprocessed as described below. We call this algorithm *partitioning into exact search with preprocessing* in the following.

We implemented this algorithm in the software library and the algorithm is available using `Partitioning<IntoExactSearch>` with `DoPreparePatterns=True`.

3.3.6.1 Similarity measure

The algorithm can be used with the same similarity measures as the PEX algorithm (Section 3.2.3) and partitioning into exact search (Section 3.3.3).

3.3.6.2 Supported index structures

The algorithm is specialized for the use with suffix forests in external memory (in the implementation it works with `IndexDigest`).

3.3.6.3 Search algorithm

The algorithm for approximate pattern matching in suffix forests consists of two phases: the *preprocessing* phase and the *query answering* phase.

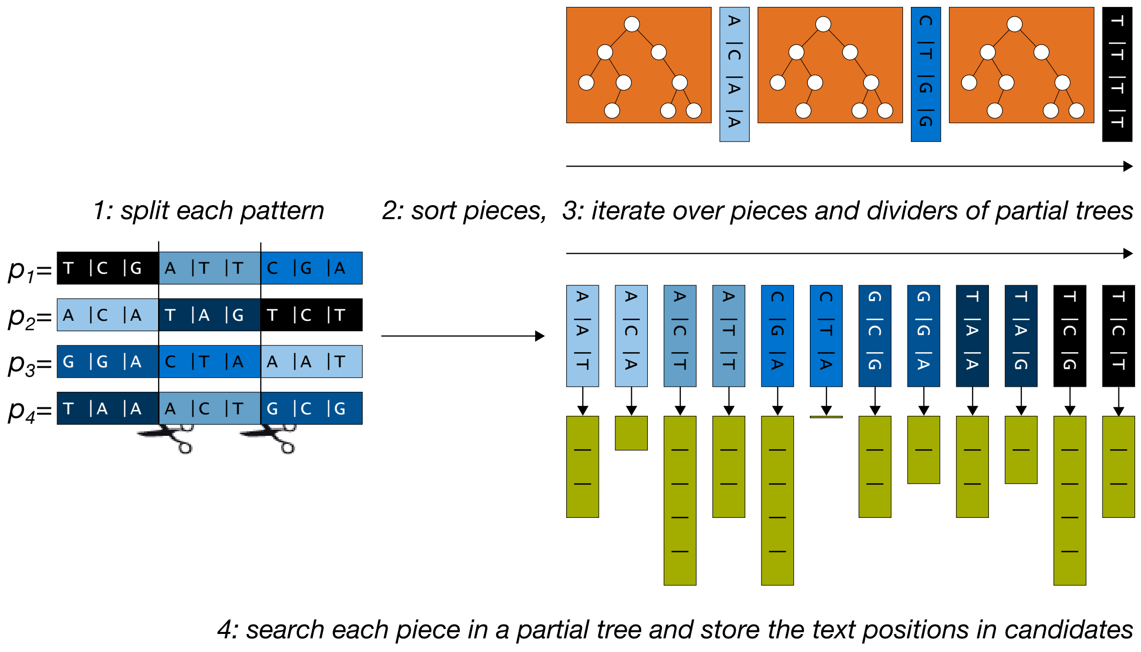

1: split each pattern
2: sort pieces,
3: iterate over pieces and dividers of partial trees
p_1= T C G A T T C G A
p_2= A C A T A G T C T
p_3= G G A C T A A A T
p_4= T A A A C T G C G
4: search each piece in a partial tree and store the text positions in candidates

Figure 15: Approximate search with partitioning specialized for suffix forests.

The *preprocessing phase* of the algorithm takes as input a *set of patterns* and performs four steps:

1. Each pattern is split into $k + 1$ pieces.
2. All pieces are sorted in ascending lexicographic order.
3. The algorithm simultaneously iterates over the sorted pieces and the sorted dividers of the partial suffix trees.
4. If a divider interval contains a piece, the corresponding partial tree is loaded from disk into main memory. All contained pieces are searched in the partial tree and any found text position is appended to the *candidates* list of the respective piece.

The *query answering phase* does not need to access the index again but uses only the information stored in *candidates*.

Preprocessing. The input of the preprocessing phase is the set of all search patterns that are going to be used in the query answering phase. Each pattern is split into $k + 1$ pieces just as in the partitioning into exact search algorithm. This yields the set of all pieces that are going to be searched in the query answering phase using exact search. All those pieces are then sorted lexicographically and duplicates are filtered out to reduce the size and to avoid redundant work. The algorithm is shown schematically in Figure 15.

The algorithm then tries find the positions of the pieces in the text with as little I/O operations as possible. It can take advantage of lexicographical ordering of the pieces because the dividers (and the corresponding partial trees) are sorted lexicographically as well. We therefore iterate simultaneously over the sorted dividers and the sorted pieces. For each interval defined by two consecutive dividers, the algorithm decides if it contains at least one piece (by using the binary prefix stored within each divider and only on equality by also comparing in the text).[14] If a piece is contained, the corresponding partial tree is loaded to main memory. Each partial tree is therefore only loaded at most once. All pieces that are potentially contained in the current partial tree are then searched using the regular exact search algorithm as described in Section 2.2.7.3 on page 46 (blindly follow the edges and on success verify for an actual match using the text). All starting positions of each found piece are finally stored in a data structure *candidates* held in main memory.

[14] Some caution has to be used because a piece can be contained in more than one tree.

Query answering. The actual query answering for one search pattern works exactly the same as partitioning into exact search, only that the index structure itself is not touched again because all exact search queries can be answered using the *candidates* data structure held in main memory. Each matching candidate position is verified using an online algorithm.

The number of patterns is denoted by l here. For the analysis of searching l patterns we assume all patterns are of the same length m and all queries use the same search tolerance k. Sorting the pieces and filtering duplicates can be achieved in $\mathcal{O}(l\,m)$ worst case time by using a generalized suffix tree and an online construction algorithm.[15] The time of the further preprocessing and the query answering phase is bounded by l times the time necessary for partitioning into exact search because our algorithm performs the same operations (or even less in case of duplicates). The *amortized running time* for one query is therefore $T_{\text{Partitioning (with preprocessing, amortized)}} \leq T_{\text{Partitioning (exact)}}$. The space usage depends linearly on the number of matching candidate positions and is therefore $S_{\text{Partitioning (with preprocessing)}} = \mathcal{O}(|candidates|) = \mathcal{O}(n)$.

3.3.6.4 Discussion

Our proposed algorithm is one step in the direction of a more efficient algorithm for approximate pattern matching in suffix forests. It can work especially well compared to the basic partitioning algorithm if the patterns are sufficiently long. For very short patterns there are many candidate positions which need to be stored in main memory ($m = 4$ with $k = 1$, e. g., leads to a piece length of 2 with presumably a very high number of candidates).

An implementation of the DiGeST index with a graphical user interface for searching was recently proposed by Minkley et al. [Min+14]. They use a backtracking algorithm in the suffix forest, but the distance measure is limited to Hamming distance and wildcards (which is substantially easier because no dynamic programming is necessary). The results of the running time indicate that this backtracking algorithm is only feasible for very small values of k.

Another recently proposed approach for approximate pattern matching in disk-based suffix trees is by Watanuki et al. [Wat+13]. They, however, do not use a partitioning of the suffix tree based on prefixes (such as, e. g., DiGeST [Bar+08] and Trellis [PZ07]), but initially split the underlying text in smaller partitions. Each query therefore has to be carried out in every resulting partial tree to find all matches in the text. The proposed algorithm therefore uses a parallelization of the backtracking on a multi-core CPU and an elaborated buffer management (the described algorithm is limited to Hamming distance).

3.3.7 Approximate search in compressed indexes

It is possible to perform approximate pattern matching in compressed index structures (Section 2.3) by simply applying the algorithm for their respective uncompressed counterpart (e. g., suffix array or suffix tree) using only the abstract interface. However, there are also specialized algorithms for the different index structures that can better exploit the internals of the index. In many cases they make use of the *bidirectionality* of those index structures (see Section 2.3 and [Ohl13]). Some of the current algorithms for compressed index structures are outlined briefly in the following paragraphs.

The compressed suffix array CSA (Section 2.3.2) can be used with an approximate search algorithm by Sadakane and Shibuya [SS01] which is based on backward searching and splitting the pattern. The proposed index and algorithm can, for example, be used to search in the human genome on a desktop computer with 2 GB of main memory (after constructing the index on a machine with more main memory) [SS01].

[15] In the implementation, however, we use a light-weight skip-list data structure for storing the pieces.

The compressed suffix array GV-CSA by Grossi and Vitter [GV00] is used by Huynh et al. [Huy+04] and Huynh et al. [Huy+06] for approximate pattern matching. They propose an algorithm for $k = 1$ using neighborhood generation, forward and backward searching, and also extend the algorithm to general values $k > 1$.
The compressed suffix tree by Sadakane [Sad07] is used by Lam et al. [Lam+08b]. The approximate search algorithm furthermore stores additional data structures inside the tree nodes. The search algorithms works for $k = 1$ and also generalized for higher search tolerances $k > 1$.
The FM index by Ferragina and Manzini [FM00] is used by Lam et al. [Lam+08a] to build the software tool BWT-SW that efficiently computes local alignments for DNA sequences in practical applications; the algorithms uses backward search to simulate a traversal over the conceptual suffix trie and dynamic programming with pruning of unnecessary subtrees. The FM index is also used by Langmead et al. [Lan+09] and Langmead and Salzberg [LS12] to build the software tools Bowtie and Bowtie 2 for aligning short DNA sequences to a long sequence (e. g., the human genome); the algorithm uses dynamic programming and backtracking and additionally heuristics to speed-up the calculation in the biological context.
The LZ index by Russo and Oliveira [RO08] is used by Russo et al. [Rus+07] to develop an approximate search algorithm. The algorithm uses a hybrid approach of splitting the pattern and backtracking in the conceptual trie of phrases. The results are stated to carry over to other LZ-based indexes
Russo et al. [Rus+09b] propose an algorithm for compressed indexes that uses hierarchical verification. If a matching piece is found it is extended to both sides allowing a higher tolerance to try to avoid costly verifications. The proposed algorithm works with all compressed indexes supporting bidirectionality (e. g., compressed suffix arrays, FM index, compressed suffix trees).

A survey by Russo et al. [Rus+09a] summarizes current algorithms for approximate search in several compressed indexes and also includes a comparative experimental study. The book by Ohlebusch [Ohl13] gives an overview of several compressed indexes and pattern matching algorithms, also for approximate search.

In the software library it is possible to perform approximate pattern matching in compressed indexes using the partitioning into exact search algorithm (Section 3.3.3).
The implementation of the FM index additionally offers a *prefix trie interface* which would make it possible to adapt our backtracking algorithm (conceptually working on the suffix tree) also for the use with this data structure.

3.3.8 Specialized indexes for approximate string matching

There are several index structures designed only or especially for approximate pattern matching. These indexes are able to solve approximate pattern matching problems faster in theory and/or in practice than the indexes presented in the previous Chapter 2. This usually comes at the price of a bigger index data structure. However, in this work we focus on indexes which are applicable more generally and which can also be used to efficiently solve other problems while still being reasonably small (see Section 1.5), so some of these specialized approaches are only briefly described here.

One general approach is to store not only text substrings (if the problem instance is a text) or dictionary entries (if the problem instance is a dictionary) but also their neighbors in the index structure. In the extreme case, all neighbors of each dictionary entry with distance $\leq k$ are stored together with a pointer to the original entry; this permits to perform approximate pattern matching of a pattern in optimal time $\mathcal{O}(m + occ)$. There are several indexes using suffix tree based data structures which are extended with such neighborhood information: This includes the index of

Cole et al. [Col+04] (*k-Errata-Trees*), the index of Maaß and Nowak [MN05a] (*weak trees/error trees*), the index of Coelho and Oliveira [CO06] (*dotted suffix trees*), the index of Tsur [Tsu10], the cache-oblivious index by Hon et al. [Hon+07b; Hon+11], and the compressed index of Chan et al. [Cha+06b] based on the index of Cole et al. [Col+04].
Another index that stores the neighborhood of the dictionary entries is FastSS by Bocek et al. [Boc+07]: Instead of storing *all neighbors* with distance $\leq k$ regarding edit distance, they only store the so-called *deletion neighborhood*, i. e., all strings that can be generated by deleting up to k characters from a dictionary entry. At query time, the deletion neighborhood of the pattern is generated and simply looked up in the index. A similar algorithm for $k = 1$ is also used by Mihov and Schulz [MS04].
The B^{ed}-tree by Zhang et al. [Zha+10] stores strings in a B^{+}-tree organized for an efficient answering of approximate pattern matching queries, including top-K-queries.

Another branch of indexes specialized for approximate pattern matching uses two index structures: one for the text and one for the reversed text (or one for the dictionary entries and one for the reversed dictionary entries). To perform an approximate search with $k = 1$, the pattern is split at every possible position and the first piece is searched in the forward index and the second piece in the reverse index. Finally, the matching positions have to be combined using additional data structures.[16] This general approach is the basis of the index of Amir et al. [Ami+00] and the index of Mihov and Schulz [MS04] (backwards dictionaries). The algorithms in their basic form work for $k = 1$ but can be extended to $k > 1$.
The advantage of those specialized index structures is that they can offer very fast search functionality in theory (even optimal time, e. g., in the index by Maaß and Nowak [MN05a]) or in practice. The disadvantage is that they need in general more space than the classical index structures which reduces their usefulness in practice. Often the search tolerance is limited to $k = 1$ or has to be constant and known in advance.

3.3.9 Metric indexes

The search of a pattern p in a dictionary D with search tolerance k regarding a distance measure δ can also be seen as the search in a metric space (if δ is a metric).
A function $\delta : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ is called *metric* if for all $p, r, s \in \Sigma^*$:

1. $\delta(r, s) \geq 0$ *(non-negativity)*
2. $\delta(r, s) = 0$ if and only if $x = y$ *(identity of indiscernibles)*
3. $\delta(r, s) = \delta(s, r)$ *(symmetry)*
4. $\delta(r, p) \leq \delta(r, s) + \delta(s, p)$ *(triangle inequality)*

Some of the distance measures for strings mentioned in Section 3.1 satisfy indeed these conditions and are therefore metrics. This includes the Hamming distance and the simple edit distance, as well as the weighted edit distance (if each operation costs > 0 and the cost matrix is symmetric). An edit distance with extended operations (e. g., character transposition, merge, split) is a metric if each operation has an inverse operation of the same cost [Nav01].
A general alignment score with positive and negative scores is not necessarily a metric because none of the four conditions above can be guaranteed.
Solving approximate pattern matching problems for a metric can then be reduced to searching a metric space. An approach that uses a metric index to search in a dictionary (or in the words of a text) was proposed by Baeza-Yates and Navarro [BYN98] for natural language texts.

[16] This problem is in our view closely related to the search in an LZ index using the forward and backward trie (Section 2.3.3.3) even though this is not mentioned.

There are generic metric index structures that organize the elements by only taking into account the pairwise distance values (and exploiting the triangle inequality). These index structures are usually trees and therefore called *metric trees*. The metric space is partitioned into smaller areas, allowing for a more efficient search than performing a brute force search. Examples of metric trees using *ball partitioning* are the BK-tree, the VP-tree[17], the MVP-tree[18] and the FQ-tree[19]. Metric trees using *hyperplane partitioning* are the BS-tree[20] and the GH-tree[21]. The book by Samet [Sam05] gives an overview of metric index structures.
The biggest advantage of using metric indexes for pattern matching is the flexibility regarding the distance measure. The main disadvantage of this approach is the running time to answer queries with higher search tolerance. Already for $k = 2$ and simple edit distance, Baeza-Yates and Navarro [BYN98] report that an online search outperforms searching the metric index (for a problem instance of size 5 MB). We therefore did not implement this approach here.

There are other approaches for using metric indexes. A method to solve pattern matching problems of texts with metric trees was proposed by Chavez and Navarro [CN02]. They index the internal nodes of the suffix tree in a metric tree. Bartolini et al. [Bar+02] use a metric index for the bag distance of the strings (based on counting how many characters the two strings have in common, regardless of their order), combined with a verification step. It is therefore reported to work especially well when the alphabet is large, but worse if it is small as, for example, in DNA sequences.

3.3.10 Top-K-queries

A problem not studied in further detail in this thesis is the top-K approximate string matching problem (recall from Section 1.4): Given a dictionary D, a search pattern p and a number K, find those K strings $s_i \in D$ that have the smallest distance to the search pattern (or the highest similarity, respectively).
A frequent approach is to start with tolerance $k = 0$ and to increase the search tolerance step by step until sufficiently many matching strings with $\delta(s_i, p) \leq k$ have been found.
Specialized solutions for the top-K matching problem have been developed, for example, by Vernica and Li [VL09] (in the field of information retrieval and also taking into account the importance of terms in the dictionary), Yang et al. [Yan+10] (using a q-gram based framework), Deng et al. [Den+13] (using dynamic programming with pruning and avoiding duplicate computations), and Kim and Shim [KS13] (using filtering with q-grams indexes).

3.4 Summary

The algorithms we implemented for approximate pattern matching using index structures are summarized in Table 4.

[17]VP = **v**antage **p**oint
[18]MVP = **m**ultiple **v**antage **p**oint
[19]FQ = **f**ixed **q**uery
[20]BS = **bis**ector
[21]GH = **g**eneralized **h**yperplane

Algorithm	Reference	Implementation	Section	Requires suffix tree iterator
`DPBacktracking`	Navarro et al. [Nav+01]	Poppe [Pop10] and Krugel [this work]	3.3.2	●
`Partitioning<IntoExactSearch>`	Navarro et al. [Nav+01]	Poppe [Pop10] and Krugel [this work]	3.3.3	○
`Partitioning<Intermediate>`	Navarro and Baeza-Yates [NBY00]	Poppe [Pop10] and Krugel [this work]	3.3.4	●
`Partitioning<PartitioningHierarchical>`	Myers [Mye94], Hyyrö et al. [HN03]	Krugel [this work]	3.3.5	○
`Partitioning<IntoExactSearch>` (with preprocessing)	Krugel [this work]	Krugel [this work]	3.3.6	○

Table 4: Summary: implemented algorithms for approximate pattern matching.

4 Software libraries

There exist many software libraries that have capabilities for (approximate) string matching. They come from different areas such as databases, information retrieval systems, bioinformatics, theoretical computer science, or operating systems and also are targeted at different types of problems. Some focus on index structures, others on pattern matching algorithms and yet other libraries focus on string similarity measures.
Here we present a selection of the in our view most important libraries for solving approximate pattern matching problems in practice. We decided to only include libraries that:

1. Solve approximate pattern matching using an index,
   or provide *several* indexes and/or algorithms,
2. Are reusable for different problems and have a clean programming interface,
3. Are open source, and
4. Come with some kind of ‘‘free’’ license.

We therefore do not include, e. g., implementations of only one index structure for exact matching, implementations without a clean programming interface, or prototypical implementations of indexes/algorithms. All described libraries are summarized in Table 5 on page 103.

4.1 Index structures for approximate search

SeqAn is a software library of algorithms and data structures for sequence analysis with special focus on biological data by Döring et al. [Dör+08]. It offers several efficient algorithms for exact and approximate online pattern matching with different string measures (including edit distance and alignments). The library furthermore contains different index structures for strings. The library is implemented in C++ and makes heavy use of template programming. It is usable as header-only library without the need to compile it first.
In addition to the core library, the project also contains applications for bioinformatics problems such as read-mapping. Recently, an integration into KNIME (Konstanz Information Miner) has been added to allow the easy creation of workflows for data analysis.

We added additional index structures for exact search (as described in Chapter 2) and algorithms for approximate searching in index structures (as described in Chapter 3), which did not exist when we started our project (in Table 5 we added, colloquially speaking, the blue square in column 4.1).

NVBIO is a software library for bioinformatics applications developed by NVIDIA Corporation [NVI14]. It is especially designed to efficiently use the parallel computing power of modern GPU (graphics processing units) by using the CUDA application interface. The library contents are similar to SeqAn, including among others index structures (suffix trie, q-gram index, FM index) and approximate pattern matching algorithms (Myers algorithm, dynamic programming, and backtracking in the FM index).

Lucene/Solr is a software library for information retrieval and full-text searching managed by Apache Software Foundation [Apa14].[1] The project includes web-crawlers, indexers, document parsers, database handlers, a standalone full-text search server and many more functions to form a complete information retrieval platform. The focus of this project lies on natural language texts

[1]The two projects have been merged recently.

which are structured as words (and not on biological sequences). When used for approximate string matching, an inverted index and several string measures (edit distance, phonetic measures, wildcards, etc.) can be applied. The library is written in Java but has bindings and is ported to several other languages.

Flamingo Package is a software library for approximate string matching by Behm et al. [Beh+10]. It is implemented in C++ and contains on the one hand index structures such as a filter tree (using a filter criterion and inverted lists) and a spacial index (mapping edit distance queries from metric to Euclidean space) and on the other hand algorithms, e. g., for list merging and answering top-K-queries.

SimString is a lightweight software library for similarity search in a collection of strings by Okazaki and Tsujii [OT10]. It uses an inverted index, supports q-gram-based string measures (Jaccard, Dice, Cosine), and also handles Unicode sequences. It is implemented in C++ as headers-only library and has bindings for Python and Ruby.

libcolumbus is another lightweight software library for similarity search in a collection of strings developed by Canonical Ltd. [Can12] for the Ubuntu operating system. It uses a trie as index structure and supports edit distance and weighted edit distance. It is implemented in C++ and also has a C and Python interface.

PATL: Practical Algorithm Template Library offers data structures for storing a set or a map of strings using tries and is implemented by Klyujkov [Kly09]. The data structures are compatible with the corresponding C++ standard template library classes. The library also offers iterators for approximate search based on Hamming or edit distance (also with transpositions and character merge/split operations).

SecondString is a project by Cohen et al. [Coh+03] and contains an implementation and comparison of many different string measures. The package is implemented in Java and also contains data structures for strings using methods from information retrieval (TFIDF). It focuses primarily on the properties of the string measures and how they can be learned from training sets. It is, however, “*not designed for use on large data sets*” [Coh+03].

4.2 Index structures for exact search

Pizza & Chili Corpus is a project by Ferragina and Navarro [FN05] and contains a collection of (mostly compressed) index structures including different versions of the compressed suffix array, FM index, LZ index, and compressed suffix tree. The indexes all have a simple unified programming interface in C with functions for construction, storing, loading, and exact searching. Additionally, the project contains a collection of test instances for performing benchmarks. A detailed description for practical applications and a comparison is given by Ferragina et al. [Fer+09a].

libcds/libcds2 by Claude [Cla08; Cla13][2] provide basic data structures that can be used, for example, in compressed indexes. This includes several efficient data structures for bit vectors, wavelet trees and permutations. The development of libcds is not continued, but the new version libcds2 is developed.

[2]libcds/libcds2 = **C**ompact **D**ata **S**tructures **Lib**rary / **C**ompressed **D**ata **S**tructure **Lib**rary

SDSL by Gog [Gog14][3] also provides basic data structures for the use in compressed indexes. This includes several efficient data structures for bit vectors and wavelet trees but also complete implementations of different variants of compressed suffix arrays and trees.

4.3 Online approximate search

String::Approx Is an extension for Perl to perform online approximate pattern matching by Hietaniemi [Hie13]. It uses edit distance and additionally permits some modifiers (e. g., maximal number of deletions for a match) to be applied.

TRE is a regular expression library written in C by Laurikari [Lau09]. It also supports *approximate* regular expressions and can therefor use the simple or weighted edit distance.

Boost.RegEx by Maddock [Mad01] is part of Boost, a comprehensive collection of libraries for C++ [Boo14]. The RegEx library contains several algorithms for searching regular expression in many different flavors. The library is quite mature, well tested and documented.

Python regex is a module for regular expressions in Python developed by Barnett [Bar14], currently in beta version. It also contains functions for *approximate* regular expressions using edit distance and additional modifiers restricting, e. g., the number of deletions.

4.4 String measures

LEDA is a software library[4] with focus on data structures and algorithms for graphs and on computational geometry. It was developed at the Max Planck Institute for Informatics Saarbrücken by Mehlhorn and Näher [MN99] and is now continued at Algorithmic Solutions Software GmbH [Alg14]. The library also contains modules for calculating string distances (Hamming, edit distance, weighted edit distance), computing alignments, and various algorithms performing online exact string matching. The library is implemented in C++ and available with three different licenses (free, research, and professional version). The string algorithms are available only in the research and professional version.

natural for Node.js is a natural language processing system written in JavaScript for the server platform Node.js implemented by Umbel et al. [Umb+11]. It contains functions to compute string measures (edit distance, Jaro-Winkler) and an implementation of a simple trie data structure for exact search.

SimMetrics is a comprehensive library of algorithms for many different string measures by Chapman [Cha06] (in total more than 30 similarity or distance measures). The library is implemented in Java, is also available for .NET, and includes a documentation and comparison of the measures. (The overview and documentation web page seems to have gone offline but the library itself is still available online.)

stringmetric is a library of algorithms for many different string measures by Madden [Mad13] (in total more than 20 similarity or distance measures). The library is implemented in Scala 2.10 (a functional and object-oriented programming language) and also offers a command line interface for the computations.

[3]SDSL = **S**uccinct **D**ata **S**tructures **L**ibrary

[4]LEDA = **l**ibrary of **e**fficient **d**ata types and **a**lgorithms

4.5 Bioinformatics

There are several other software libraries for bioinformatics (called, e. g., *Bio++*, *BioJava*, *BioJS*, *BioPerl*, *BioPython*, *BioRuby*, *BIU*, *BTL*, and *Libcov*). They typically have their focus not so much on data structures and algorithms for pattern matching but on more biological aspects like phylogenetic problems and statistical analyses.

4.6 Summary

The software libraries of index structures and algorithms for pattern matching are summarized with their provided functionality and license in Table 5.

Name	Version[6]	Reference	Language	Offered Functionality [5]				License
				4.1	4.2	4.3	4.4	
SeqAn	2.0.0 (2015-02-11)	Döring et al. [Dör+08]	C++	🟦 ←	●	●	●	BSD 3-clause
NVBIO	1.1.50 dev (2015-02-13)	NVIDIA Corporation [NVI14]	C++	●	●	●	●	BSD/GPL 2
Lucene/Solr	5.0.0 (2015-02-20)	Apache Software [Apa14]	Java & more	●	●	○	●	Apache License 2.0
Flamingo	4.1 (2012-02-22)	Behm et al. [Beh+10]	C++	●	●	○	●	BSD, Academic BSD
SimString	1.0 (2010-03-07)	Okazaki and Tsujii [OT10]	C++ & more	●	●	○	○	BSD 2-clause
libcolumbus	1.0.0 (2013-10-18)	Canonical Ltd. [Can12]	C++ & more	●	●	○	○	LGPL 3
PATL	Rev. 129 (2010-08-03)	Klyujkov [Kly09]	C++	●	●	○	○	BSD 2-clause
SecondString	Alpha (2012-06-20)	Cohen et al. [Coh+03]	Java	●	○	○	●	NCSA Open Source
Pizza & Chili	[7]	Ferragina and Navarro [FN05]	C	○	●	○	○	LGPL 2.1[8]
libcds/libcds2	2.0 (2014-07-23)	Claude [Cla08; Cla13]	C++	○	◐	○	○	LGPL 2.1
SDSL	2.0.1 (2013-11-05)	Gog [Gog14]	C++	○	●	○	○	GPL 3
String::Approx	3.27 (2013-01-22)	Hietaniemi [Hie13]	Perl	○	○	●	●	LGPL 2, Artistic Lic. 2.0
TRE	0.8.0 (2009-09-20)	Laurikari [Lau09]	C & more	○	○	●	●	BSD 2-clause
Boost.RegEx	5.0.0 (2014-09-25)	Maddock [Mad01]	C++	○	○	●	●	Boost Software License 1.0
Python regex	Beta (2015-03-18)	Barnett [Bar14]	Python	○	○	●	●	Python Software Foundation
LEDA	6.4 (2012-07-09)	Algorithmic Solutions [Alg14]	C++	○	○	○	●	free/research/professional
natural for Node.js	0.1.26 (2014-02-05)	Umbel et al. [Umb+11]	JavaScript	○	○	○	●	MIT
SimMetrics	1.6.2 (2007-02-07)	Chapman [Cha06]	Java	○	○	○	●	GPL 2.0
stringmetric	0.27.4 (2014-07-02)	Madden [Mad13]	Scala	○	○	○	●	MIT

Table 5: Software libraries for pattern matching.

[5] The numbers refer to the sections of this chapter:
4.1 Index structures for approximate search
4.2 Index structures for exact search
4.3 Online approximate search
4.4 String measures

[6] The version numbers and dates of the last release/update are as of April 2015.
[7] Each implemented index has its own version number.
[8] The different implementations come with individual licenses but nearly all use the LGPL 2.1.

Part III

Evaluation

5 Test instances

A basic ingredient for systematic experimental evaluations is a set of test instances. In our case, we need texts of different kinds and sizes to evaluate algorithms for approximate pattern matching. We decided to use on the one hand *real world* texts to test the algorithms under *realistic* conditions, and on the other hand *synthetically generated* texts to test them under *controlled* conditions.

We collected real world texts of different types (including natural language texts and DNA sequences) and preprocessed them to be usable as input for the pattern matching algorithms (Section 5.1).
The performance of pattern matching algorithms depends heavily on the statistical properties of the underlying text. We therefore implemented tools to analyze a text regarding several statistical properties, including simple properties such as the text length and the alphabet size and more complex properties such as the distribution of q-grams and the entropy (Section 5.2).
We furthermore implemented text generators that output synthetic sequences using different models: a simple Bernoulli process, a Fibonacci sequence generator, a stochastic process using Markov chains, and an extension that explicitly models repeated substrings (Section 5.3). The parameters of the stochastic processes can be learned from training data. This enables us to provide synthetically generated texts which are similar to real world texts but furthermore have very controlled properties.

Problem instances for approximate pattern matching consist not only of a text but also require a set of *search patterns*. We therefore implemented a search pattern generator (Section 5.5).

5.1 Real world test instances

Most applications of approximate pattern matching either handle natural language texts or biological sequences (see Section 1.1), so we decided to use these two general types of texts in our real world test sets. To cover a broad range of realistic scenarios we decided to include natural language texts of different languages (English, German, and Chinese) and biological sequences of different origins (DNA and proteins sequences).
For each type we downloaded files from different sources and performed several preprocessing steps to clean up the test sequences and to achieve a unified format. This includes the conversion of new-line characters, the harmonization of character encodings, and the removal of header and/or footer information to retain only the actual sequence data. This results in a set of (rather small) texts for each type. We concatenate many of those individual texts to form a reasonably long actual test instance. We decided to maintain information about the small texts in a relational database which enables us to select a subset of texts that matches certain criteria. This makes it possible for us to select only texts from a certain origin (e. g., only DNA sequences of the human genome or only German texts) and with chosen statistical properties (e. g., only texts having a big alphabet or a low entropy).

5.1.1 Natural language texts

For the natural language texts, we decided to download publicly available books from the Project Gutenberg [Har71] website. We used a download robot and retrieved the text files in August 2009. We unzipped the files and deleted duplicates of the same book (which were detected using the file names). The individual books use different character encodings, and since no additional meta-information was available we had to guess the encodings from the *file contents*; we did this

using the Linux `file` command (which, however, only works heuristically and recognized some files as plain ASCII even if they use a some non-ASCII characters). All new-line characters were converted to `\n` (using the Linux `sed` command). Additionally, we wrapped all lines longer than 300 characters, so that all files are in a unified format.
Each book's text file includes a header and/or a footer containing meta information about the book, as well as a license of Project Gutenberg. In order to comply with this license when providing the texts ourselves for download, we had to remove all references to Project Gutenberg. Another reason for stripping the license text is to avoid the unrealistically frequent repetition of very similar text blocks. Unfortunately, the headers and footers have no standardized format and so we implemented a tool to strip these elements by using several heuristics regarding the occurring text fragments (successfully removing the headers and footers of all files).

Texts in different natural languages differ heavily in their statistical properties, not limited to but very prominently visible in the alphabet size: Latin-based texts have an alphabet size of less than 200 (including lower and upper case letters, numbers, and punctuation), while many other, e. g., Asian languages have alphabets with several thousand characters. We therefore collected English and German texts, as well as Chinese texts with a total file size of about 10 GiB. An example text file can be found in the Appendix (Section A.2.1).

5.1.2 DNA sequences

Several prominent applications of pattern matching work on DNA sequences and so we also compiled DNA sequence test instances. We downloaded DNA sequence files from the NCBI[1] database *GenBank* [Nat09] in the FASTA file format[2] in August 2009. As part of the preprocessing, we split the FASTA file so that there is one file per biological sequences (e. g., one file per chromosome).
A DNA molecule consists of a long chain where each element is one of the four nucleotides Adenine, Cytosine, Guanine, and Thymine. It can therefore be represented using a string over the alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. However, the DNA sequence files additionally contain other characters that model uncertain data. Each of these additional characters represents a so-called *character class* of the IUPAC code [CB84], i. e., a subset of the alphabet (e. g., $R = \{\mathrm{A}, \mathrm{G}\}$ and $N = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$). This uncertainty exists for several reasons, among others because the biochemical sequencing processes cannot always unambiguously determine the occurring nucleotide at each position or due to genetic variations among individuals. The downloaded concatenation of the chromosomes of the human genome, for example, contains consecutive stretches of N with length > 100.000.
Since some pattern matching algorithms and indexes depend on the fact that a DNA sequence only contains the four standard characters, we provide the DNA sequences in two versions: One with IUPAC character classes, and another version where all those ambiguous characters are stripped out as done by Barsky et al. [Bar+08].[3]
An excerpt of an example DNA file in FASTA format can be found in the Appendix (Section A.2.2). For an easier and unified handling of the test sequences we decided to convert the files from FASTA format to plain text format by concatenating the sequences and leaving out the headers. We collected DNA sequences with a total file size of 37 GiB but for our experimental evaluation we only use the chromosomes of the human genome with a total size of 3 GiB.

[1] NCBI = **N**ational **C**enter for **B**iotechnology **I**nformation

[2] The FASTA file format is very popular in bioinformatics and stores the actual biological sequences together with meta information.

[3] To reproduce the position of matches in the original string it is possible to store some small additional information about the positions of the stripped-out characters as noted by Barsky et al. [Bar+08].

5.1.3 Protein sequences

Another popular application of pattern matching is searching in protein databases and we therefore decided to also use protein sequences as test instances. We downloaded the complete content of protein sequences from the NCBI GenBank [Nat09] in October 2010 using the FASTA file format. We split the content of the downloaded files so that each protein sequence is stored in an individual file.
For the concatenation we converted the files from FASTA to plain text format, retaining only the actual sequence data and discarding the sequence headers.

Proteins consist of 23 amino acids but the files additionally contain some wild-card characters, resulting in an alphabet size of 26 which therefore lies between DNA sequences and natural language texts. Apart from the practical importance of pattern matching in protein sequences, this statistical feature makes them especially interesting here, since it allows to measure the impact of the alphabet size on the performance of algorithms. An excerpt of an example protein sequence file in FASTA format can be found in the Appendix (Section A.2.3). We collected protein sequences with a total file size of about 7 GiB.

5.2 Text analysis

Analyzing the statistical properties of sequences plays an important role in bioinformatics as well as in areas such as linguistics, automatic translation, text compression and others. The wide range of properties that are analyzed includes the character distribution, different measures of entropy, the number of different q-grams, repeat probabilities, and many more. This statistical analysis can, for example, lead to biological or linguistic conclusions about the underlying sequence.
However, our main motivation for the statistical analysis of texts is to enable a systematic comparison of data structures and algorithms for strings. The space consumption and the running time of index structures and algorithms for pattern matching depend on the statistical properties of the underlying text: The size of a q-gram index, a suffix tree, and a compressed index depends, for example, on the number of different q-grams, the number of repetitions, and the entropy of the text, respectively (these dependencies are explained in more detail below). The index structures can perform very differently depending on these properties, even for texts of exactly the same length.
We therefore designed and implemented a text analysis tool and used it to compute the statistical properties of our test instances. This enables us to measure the impact of these properties on the performance of the algorithms and data structures. The text analysis tool has partly been implemented during a student project by Dau [Dau10] and was published in Dau and Krugel [DK11a; DK11b].
In the following sections we describe the implemented statistical measures. For each measure we discuss possible applications, the relation to indexes and algorithms for pattern matching, and describe how it is computed in the implementation.

5.2.1 Length

One very simple but important property of a text is its length. We have to be careful, however, because we have to distinguish between three different measures here:

1. The *sequence length in characters* is the number of characters of the sequence.

2. The *sequence length in bytes* is the number of bytes used to store the sequence. Compared to the sequence length in characters, this measure is *the same* for ASCII and

ISO-8859 based encodings, *higher* for UTF-8 encoding (because several bytes might be necessary to store one character), and *lower* if several characters are stored together in one byte.

3. The *file size in bytes* consists of the sequence length in bytes plus the size of any meta information which is stored in the same file but is ignored when performing pattern matching (such as FASTA headers).

The performance of most indexes and algorithms for pattern matching depends heavily on the length of the text. Often this dependence is linear as, for example, the search time of many online algorithms and the space consumption of many indexes.

To compute the *sequence length in characters* we use the file size in bytes for files with single-byte encodings (e. g., ASCII) and without any meta information. Otherwise (when using, e. g., a variable multi-byte encoding such as UTF-8), we have to iterate over whole text to count the characters.

5.2.2 Alphabet

A text string is defined as a sequence of characters drawn from a finite alphabet Σ (see Section 1.2). In practice, the alphabet depends on the application: the alphabet of DNA sequences is usually the set $\{\,\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\,\}$ of size 4 and for natural language texts is often represented by one byte of length 8 bit and therefore of size 256.
The size of the alphabet plays an important role for several index structures and algorithms: The size of the trie or suffix tree (Section 2.2) or *q*-gram index (Section 2.4) can grow exponentially with the alphabet size and algorithms for approximate search are expected to perform very differently for varying alphabet sizes (as discussed in Chapter 3).

The *actually used alphabet* of a text is sometimes more interesting than the alphabet from which the characters are drawn (which is larger if not all characters are used). We therefore compute the actually used alphabet for all test instances. This is done by linearly scanning the text and recording all occurring different characters.

We additionally compute another measure related to the alphabet size, namely the *inverse probability of matching* [FN05]. It is the reciprocal of the probability that two randomly chosen text characters match. This measure can be used as an indication of the size of the effectively used alphabet. For uniformly distributed texts, it equals the alphabet size [FN05]; a character that only occurs once in a long text is, for example, nearly ignored by this measure.

5.2.3 Distribution of *q*-grams

Analyzing the alphabet size of a given text only gives information about single characters. It is, however, also possible to extend this and to analyze the occurrences of several consecutive characters using *q*-grams. The *q*-grams of a text are the consecutive substrings of length $q \in \mathbb{N}$ (as defined in Section 2.4). An interesting statistical measure is the number of different *q*-grams of a given text for varying values of *q*. The number of different 1-grams, for example, is the same as the size of the actually used alphabet, and the number of different 5-grams of a text gives information about substrings of length 5 (usually not all such possible substrings actually occur in a given text).
Instead of only counting the *number* of different *q*-grams, the actual *distribution* is sometimes also of interest. This distribution can be represented by a table storing for each distinct *q*-gram the number of occurrences.

In bioinformatics, the distribution of q-grams is investigated to draw biological conclusions and is, for example, compared among different species [Cho+09] or within coding regions of the human DNA [Bal+95]. Another well known application of q-gram analysis is the difference of the "CpG" content[4] in coding and non-coding regions, respectively [Fat+05; Cho+09]. Information about the distribution of q-gram plays furthermore a central role in genome assembly algorithms as well as in sequence alignment algorithms [MK11].
In natural language texts, q-grams can, for example, be used to identify the language of a given text [Dun94].
The number and also the distribution of q-grams plays a role for the performance of pattern matching algorithms, especially for the q-gram index but also for filtering algorithms such as the partitioning algorithm (Section 3.3.3).

We compute the q-gram distribution of a text by linearly scanning the text with a window of size q and recording the occurring q-grams in a map structure. We implemented a few variants to also handle larger alphabets, differing in how the q-grams are stored (storing the q-grams as strings or only pointers to starting positions in the text) and which data structure for the map is used (tree-based map, hash map, etc.).
Going beyond this, there are more advanced solutions to compute q-gram distributions of large texts if efficiency plays a major role, e. g., using parallel algorithms as by Marçais and Kingsford [MK11].

5.2.4 Entropy

The entropy is a measure for the randomness of a sequence or of a stochastic process. It can also be seen as a measure for the information content and provides a lower bound for certain classes of compressors. There are several variants of entropy measures, here we focus on the *empirical entropy* of a text and the *Shannon entropy* of a stochastic process (which can be estimated by analyzing sequences generated by the process).

The *empirical entropy* of order $g \in \mathbb{N}$ of a text t is written as $H_g(t) : \Sigma^* \to \mathbb{R}$ and can be used as a measure of randomness of a given text. It is defined solely on the text t itself and does not need any assumptions about the process that generated the text. It is defined for order 0 and for order $g > 0$ (also called *higher-order empirical entropy*) as described by Manzini [Man01]:

$$H_0(t) := -\sum_{u\in\Sigma} \frac{n_u}{n} \log \frac{n_u}{n}$$

$$H_g(t) := \frac{1}{n} \sum_{r\in\Sigma^g} |t_r|\, H_0(t_r)$$

Thereby denotes $n_u \in \mathbb{N}$ the number of occurrences of character u in the text t. The sequence $t_r \in \Sigma^*$ is the concatenation of all characters w that precede an occurrence of the string r.

The 0-order empirical entropy is highest ($H_0 = \log\sigma$) if all characters have the same relative frequency [Hon+03], e. g., generated by a Bernoulli process (see Section 5.3.1). It always holds $H_g(t) \leq \log\sigma$ [FM05]. The empirical entropy of a text[5] provides a lower bound for the compressibility of the text: The value $nH_g(t)$ is a lower bound for any compressor that encodes each symbol by only considering the g immediately preceding symbols [FM05].
In our text analysis tool, we compute the order-g empirical entropy of a text by collecting the strings t_r and calculating their 0-order entropy, following the definition by Manzini [Man01].

[4]The 2-gram "CG" is usually called "CpG" in bioinformatics to avoid confusion with the base pair $\mathrm{C} - \mathrm{G}$ where C is on the one strand of the DNA and G is on the other.

[5]The empirical entropy could actually also be called empirical entropy *rate* because its value refers to the information content of a single character and not of the whole text.

The *Shannon entropy* is a measure for the randomness of a random variable X_y or a stochastic process $(X_y)_{y\in\mathbb{N}}$ and is written as $H(X_y)$ [Sha+48].[6] In our context, we want to measure the entropy of a given text and we do not know the underlying stochastic process (we do, for example, not know the process which generated the DNA of the human genome). We can, however, assume that a given text was generated by a stochastic process $(X_y)_{y\in\mathbb{N}}$, i. e., in each step the process outputs a character based on the distribution of the random variables X_y. If we make some simplifying assumptions about the process, we can estimate its entropy by analyzing a sufficiently long sequence that was generated by the process. To do so, we begin by giving several necessary definitions [HD06].
The entropy H of a random variable X_y is formally $H(X_y) := -\sum_{u\in\Sigma} p(X_y = u)\log p(X_y = u)$ and represents the uncertainty about the value of the random variable X_y. It can canonically be extended to multiple random variables by using tuples of random variables. The *conditional entropy* of order g is denoted by $H(X_y \mid X_{y-g}, \ldots, X_{y-1})$ and represents the remaining uncertainty about the value of the random variable X_y, given that the values of the random variables $X_{y-g}, \ldots, X_{y-1}$ are known. The *block entropy* of order g of a stochastic process $(X_y)_{y\in\mathbb{N}}$ is defined as $H_g(X) := H(X_1, \ldots, X_g)$. The *entropy rate* $h(X)$ of a stochastic process X gives a value for the randomness of a single character and is defined using the limit value: $h(X) := \lim_{n\to\infty} \frac{1}{n} H_n(X)$.
If we want to estimate the entropy of a generating stochastic process by analyzing one sequence generated by the process, we have to make some assumptions about the process. In particular, we have to assume the process is *stationary* and *ergodic*.[7] We measure the block entropy and the conditional entropy for a given order g by using relative $(g+1)$-gram and g-gram frequencies as estimates (using the computation described in the previous section). However, this estimation for the entropy is biased resulting in too small values, since relative frequencies are used instead of actual probabilities of the (unknown) underlying stochastic process [SG96; Sch04]. There are several possibilities to accommodate for this effect. We use a correction term based on a Taylor series of order 1 as described by Schürmann and Grassberger [SG96]. Our tool for the estimation of the Shannon entropy has been published in Dau and Krugel [DK11a; DK11b], the main work has been done in a student project by Dau [Dau10].

There is a close connection between the empirical entropy and the Shannon entropy. The empirical entropy rate of order g of a text t equals for $n \gg g$ the conditional Shannon entropy of order g of the associated stochastic process when using the relative frequencies as probabilities. For order $g = 0$, this follows from immediately from the definitions; the proof for higher orders is omitted here.

In biological applications, entropy measures can be used to draw conclusions about biological sequences [Her+94; HD06]. In some species the entropy is observed to be lower in *non-coding regions*, whereas in other species it lower in *coding regions*: An analysis of the genome of Escherichia coli revealed, for example, that the entropy in coding regions is slightly lower than in non-coding regions, and the entropy of triplets of nucleotides in the correct reading frame is significantly lower than in the wrong reading frame [Lau+92]. Opposed to this, another study on the entropy of 37 eukaryotic sequences from GenBank observed a lower entropy in the non-coding regions [Man+94]. In any case, depending on the species, the analysis of the entropy of a DNA sequence can possibly give an indication when searching for coding regions of previously unidentified genes.
The entropy also plays an important role in the analysis of data structures for strings, since many

[6]We use the same symbol as for the empirical entropy because the measures are very related; they can be distinguished by the type of the argument.
[7]This is a simplifying assumption since, e. g., in genomic sequences the probability distribution changes in different regions.

compressed index data structures (like the different versions of the compressed suffix arrays, the FM-index or Ziv-Lempel based index structures, see Section 2.3) depend on the empirical entropy of the text as a factor in their space usage. The index structures therefore need less space if the text has a low entropy, which is fortunately often the case for real world instances. We focus on the *empirical entropy* (instead of the *Shannon entropy*) for analyzing the test instances because it is defined directly on the text without any assumptions about the generating process.

5.2.5 Compressibility

The compressibility of a text is theoretically limited by a lower bound depending on its entropy (see previous section). However, to also assess the practically achievable compressibility, we decided to use different state-of-the-art compressors. When compressing a text of length n using a compressor C, the *compressed size* is defined as the length of the resulting sequence. The *relative compressed size* is defined by the term $\frac{\text{original size}}{\text{compressed size}}$. We use the following three tools:

- `gzip` (GNU zip) using the Lempel-Ziv algorithm (LZ77) and Huffman coding [Deu96]
- `bzip2` using the Burrows-Wheeler transform and Huffman coding [Sew00]
- `xz` using the Lempel-Ziv-Markov chain algorithm (LZMA) [Col09]

Each tool offers a numeric option to choose a trade-off between *fast* compression (option `-1`) and *best* compression (option `-9`). For analyzing our test instances we use all three tools, each with all nine options to get a good picture of the practical compressibility.

5.2.6 Repeat structure

Many real world sequences contain correlations that can not be modeled by only considering a few consecutive characters. Such correlations can be based on repeated substrings, some kind of underlying grammar, or other so-called *secondary* or *higher level structures*.
DNA sequences are highly structured and the structural analysis of DNA sequences plays an important role in trying to understand the functioning of DNA [Bab11]. The structure is in particular based on repeated substrings (called *repeats*) which differ not only in the length of the repeat (between 2 and several thousand base pairs), and the number of repetitions but can also be reversed or complemented (A and T as well as C and G are exchanged). There are many different types of repeats, distinguished based on their statistical characteristics or biological function.[8] Substrings are often not only repeated exactly but with some modifications due to mutations or sequencing errors.
Natural language texts also exhibit a higher level structure. The structure is based on repeated phrases or set expressions and also on the underlying grammar of the language. Here we focus on substrings which are repeated with small modifications due to, e. g., inflections.
The repeat structure of a string heavily influences the performance of algorithms and data structures, such as suffix trees where the internal nodes correspond to the substrings occurring more than once [Gus97]. Repeats also have an influence on the quality of biological algorithms [Mye99a]: fragment assembly in the shotgun sequencing process, for example, works best if there are few and short repeats.

To analyze the repeat structure of a string we first have to establish a model. Standard Markov chains cannot be used here as a generating model because long range dependencies cannot be modeled using Markov chains as noted, e. g., by Wood et al. [Woo+11].[9] Modeling repeats is

[8]Among others: Satellite DNA, STR (Short Tandem Repeats), SSR (Simple Sequence Repeats), SSLP (Simple Sequence Length Polymorphism), LTR (Long Terminal Repeat), SINE (Short Interspersed Nuclear Element), LINE (Long Interspersed Nuclear Element)

[9]More details on Markov chains for sequence generation can be found in Section 5.3.3.

especially not trivial if one also wants to allow approximate repeats, i. e., repeats containing a certain number of deviations as described above.

The model by Allison et al. [All+98] is based on a Markov process combined with ideas of Lempel and Ziv. Besides generating characters according to the Markov model, there is an additional possibility to start a repeat. A *repeat* in this model is basically just a copy of a previously generated string but within a repeat there are four possible operations: copy, change, insert and delete a single character. This allows to model small deviations (just as in the simple Levenshtein distance for strings, Section 3.1.2). The parameters of the model are (apart from some technical parameters and the parameters of the additionally used Markov process which are described below in Section 5.3.3):

- the probability to start a repeat,
- the probability to insert a single character within a repeat,
- the probability to delete (skip) a single character within a repeat,
- the probability to change a single character within a repeat, and
- the probability to end a repeat.

These parameters can also be learned from a training sequence. Given a sequence, the goal is to extract the parameters of the underlying (unknown) model that generated the sequence. The resulting parameters allow to make general statements about the frequency of the approximate repeats and the degree to which the repeats differ from each other within this sequence. This can, for example, help to determine if a given sequence contains many small repeats or a few long repeats [All+98]. Other applications of the same model are by Stern et al. [Ste+01] and Dix et al. [Dix+07].

The estimation procedure works by defining a repeat graph (see [All+98] for an illustration) which represents possible explanations of how the training sequence might have been generated. The likelihood that a given sequence was generated by a model is improved iteratively by an expectation-maximization (EM) algorithm. The algorithm has per iteration a running time quadratic in the length of the sequence and needs linear space. It is possible to speedup the process by only inspecting the most relevant parts of the graph around exact repeats of a minimum length [All+98]. However, the running time still remains a limiting factor due to the quadratic growth.

This model and the parameter estimation procedure (including the speedup) have been implemented during a student project by Dau [Dau10] and were published in Dau and Krugel [DK11a; DK11b]. In the original proposal of Allison et al. [All+98] a Markov chain of order 0 or order 1 is used with two kinds of repeats, namely forward and reverse-complementary repeats. Dau [Dau10] extended this by using a *higher order Markov chain* and additionally *reverse* (non complementary) repeats.

Due to the quadratic dependence of the running time on the text length it is, however, not possible to use this model for text analysis of texts longer than $\approx 10^5$. We therefore do not use it for analyzing longer test instances.

5.2.7 Other measures for long-range correlations

There are uncountable more statistical measures for texts; we briefly describe a few approaches for long-range correlations.

The *mutual information function* allows to measure long-range correlations in texts and is used especially to analyze DNA sequences. While being computationally feasible (compared to using a Markov process of high orders) it allows to draw conclusions about the repeat structure, periodic patterns, and the distinction of coding and non-coding DNA regions [LK92; HG95; Gro+00; Hol+03]. (The mutual information function is implemented in our text analysis tool by Dau

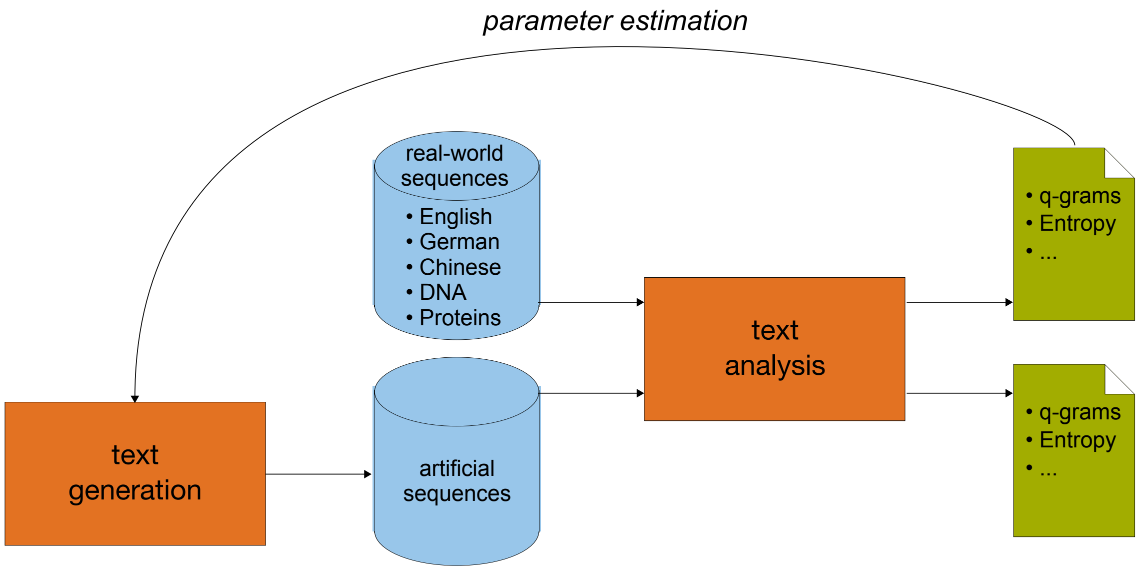


Figure 16: Work-flow of the text generation, analysis and parameter estimation process.

We use the text analysis tool to extract statistical parameters from real world test instances. The results are used to estimate the parameters of the assumed underlying stochastic process. This process is then used to generate synthetic test instances which again are analyzed for their statistical properties. These properties are then compared to those of the original sequences.

[Dau10] but we do not further investigate it here, since we do not expect a significant impact on the performance of pattern matching algorithms.) Another approach of modeling long-range correlations in DNA sequences uses a random walk model [Pen+92].

A very general model for sequences is the *Sequence Memoizer* proposed by Wood et al. [Woo+11] which tries to describe long-range correlations by using a Bayesian model while being computationally feasible.

To analyze the repeat structure of a string, a *suffix tree* can also be a useful tool since it implicitly stores repeated substrings together by representing them as one internal node (see Section 2.2.2).

There are a few tool to synthetically generate *genome-like* sequences (described below in Section 5.3.5). Some of them can learn their parameters from a given input sequence which can also be seen as an analysis of the sequences. However, completely understanding the genomic structure of, e. g., the DNA sequence of the human genome is a difficult task and not assumed to be accomplished in the near future [Int04].

5.3 Text generators

While the real world test instances allow to experiment under *realistic conditions*, it is also desirable to provide very *controlled conditions* for the experiments. We therefore designed and implemented a text generator that creates synthetic test instances using different generating models: a Bernoulli generator, a Fibonacci string generator, a Markov process, and an approximate repeats generator. The implemented methods were previously used in bioinformatics mainly to analyze sequences but not to generate them (e. g., [Deh+03; Dix+07]).

The generators have adjustable parameters that directly or indirectly influence the statistical properties of the created texts. It is therefore possible to experiment with series of parameter values and to measure the impact of the parameter on algorithms (here: on the performance of pattern matching algorithms). The parameters can be adjusted independently and therefore allow to study the impact of one parameter in isolation.

The pure synthetic generators yield artificial results which can have properties that are quite different from real world texts. While this is desirable in certain circumstances, it is also useful to have synthetic instances which are similar to real world instances with respect to certain properties. We therefore provide methods to learn the parameters of the Markov process and the approximate repeats generator from given texts. This allows to generate synthetic data with controlled, yet realistic properties. We studied the quality of this learning process by comparing the generated sequences to the original sequences and observed that some statistical properties are modeled well by the respective generating processes, whereas others are not (see below).
The work-flow of the sequence generation, analysis, and parameter estimation is sketched in Figure 16. The text generator was published in [DK11a; DK11b] and the main work has been done by Dau [Dau10].

5.3.1 Bernoulli text generator

The most simple implemented text generator outputs text sequences using a Bernoulli process. The characters are chosen from a given alphabet Σ with a uniform probability distribution and independently of the previous choices. The alphabet and the length of the strings can be defined as parameters.

5.3.2 Fibonacci string generator

A Fibonacci string (also called *Fibonacci word*) is a string over a binary alphabet (e. g., $\Sigma = \{0, 1\}$). It is defined very similarly to the Fibonacci *numbers*, only that concatenation of strings is used instead of summation. The ith Fibonacci string is denoted by f_i and defined as follows [Lot97]:

$$\begin{aligned} f_0 &= \text{``0''} \\ f_1 &= \text{``1''} \\ f_i &= f_{i-1} \circ f_{i-2} && \text{for } i \geq 2 \end{aligned}$$

The 6th Fibonacci string is, for example, the string $f_6 = \text{``}\overbrace{10110101}^{f_5}\overbrace{10110}^{f_4}\text{''}$ of length 13. The ith Fibonacci string f_i has length F_i where F_i is the ith Fibonacci number (with $F_0 = 1, F_1 = 1$). Long Fibonacci strings have many interesting properties and are in particular highly repetitive, see, e. g., [Ili+97]. They are therefore are often cited as worst-case examples for string matching algorithms and used as test instances for experimental evaluations [Gie+03; SS03; SS07; Wee12].
The implementation of the Fibonacci string generator has parameters for choosing the alphabet and the length of the string; if the length does not equal a Fibonacci number, the resulting string is a prefix of the next sufficiently long Fibonacci string. For generating the repeats of the Fibonacci string, either an in-memory buffer or a file can be used in our implemented tool.
An example of a longer Fibonacci string is in the Appendix (Section A.2.4).

5.3.3 Markov process

Markov processes are stochastic processes which are – informally speaking – memoryless, i. e., the transitions do not depend on the full history of the process but only on the previous state (or a few previous states). Markov processes have been studied in many areas and are also very popular in bioinformatics [HD06]. Of special interest are discrete-time Markov chains $(X_y)_{y\in\mathbb{N}}$ with a finite state space and which furthermore are time-homogeneous, i. e., the transition probabilities do not depend on the current point in time y. A Markov process of order g can be defined by a starting distribution for the first g symbols and a single transition probability distribution for all X_y with $y > g$. The defining property of a Markov process of order g is that the outcome of a random variable X_y only depends on the outcome of the previous g variables. In the context of sequences this means that the distribution at a certain position depends only on the g preceding characters.

The parameters of our Markov process generator are the alphabet, the starting distribution, the transition probabilities and the length of the generated sequence. In order to train a Markov process from a given sequence and to estimate the parameters by analyzing the sequence, we have to make three assumptions. We assume that the underlying Markov process is time-homogeneous, irreducible, and stationary: *Time-homogeneity* allows us to estimate transition probabilities by relative transition frequencies. *Stationarity* ensures that the starting distribution is a stationary distribution. *Irreducibility* ensures that this stationary distribution can be estimated by g-gram frequencies. The transition frequencies and the relative g-gram frequencies can be calculated in a single pass over the text counting $(g + 1)$-grams (as described in Section 5.2.3). The number of parameters of the estimation procedure grows with Σ^{g+1} and therefore only works for moderate orders g depending on the alphabet size.

Markov processes are known to accurately model *short-range correlations* between positions within a distance of at most g [HD06]. In the analysis of mammalian and non-mammalian DNA sequences it has, for example, also been observed that lower order Markov processes can also model complex q-gram distributions well [Cho+09]. However, *long-range correlations* can not be modeled well using Markov models due to the exploding number of parameters for higher orders g [Deh+03; Woo+11]. Long-range correlations are therefore considered only in our approximate repeats model (Section 5.3.4).

We examine the effect of the parameter estimation on the properties of the resulting sequence compared to the original sequence in Section 5.4. To get an impression of a synthetic text generated by a Markov process trained on an English text we provide an example in the Appendix (Section A.2.5).

5.3.4 Approximate repeats model

Another implemented text generator explicitly models repetitions of substrings by using the approximate repeat model by Allison et al. [All+98] discussed above (Section 5.2.6).

The text generation works as follows: A sequence is generated by using a Markov chain; the original proposal uses a Markov process of order 0 or 1 and we extended this to a higher order Markov process of order g [DK11a]. During this generation, a repeat is started with given probability, and the starting position of the repeat in the previously generated sequence is chosen uniformly under all preceding positions. The characters are copied starting from the chosen position; during this process, individual characters can be inserted, deleted, or substituted. The repeat is terminated with a given probability at each position (so that the lengths of the repeats follow a geometric distribution).

The parameter of the approximate repeats generator are the length of the sequence, all parameters necessary to describe the underlying Markov process, and all parameters of the approximate repeats (listed in Section 5.2.6).

Since the parameter estimation can only be executed for texts up to length $\approx 10^5$ due to memory restrictions, we do not use this model for generating test instances in this thesis.

5.3.5 Other text generators

There are many more approaches to synthetically generate texts with certain properties.

We also implemented a *discrete autoregressive process (DAR)* of order g in a student project by Dau [Dau10] which is essentially a simplified Markov process [JL83] of order g. It can be represented very compactly (with $g + \sigma + 1$ parameters) and can therefore be used also with higher orders as compared to a classical Markov process. The behavior is similar to a Markov chain of order zero but additionally has a certain probability to copy one of the g preceding characters according to a given distribution. A discrete autoregressive process models simple correlations up to order g like, e. g., the mutual information function [DK11a; DK11b]. We do not expect a close relation to the performance of pattern matching algorithms and therefore do not use this generator here.

There are several approaches to generate sequences which are similar to genomic DNA sequences. At the time when the human genome had not yet been fully sequenced, Myers [Mye99a] built a dataset generator called *celsim* for shotgun sequencing. It models the repeat structure of genomic DNA by using a stochastic formal grammar. Experiments comparing the synthetically generated sequences with real genomic sequences indicate that this model is a good approximation for the some properties of the repeat structures. However, other properties such as the entropy or the distribution of the q-grams are not modeled explicitly.
Other tools have been developed to simulate the evolution of genomic sequences: *sgEvolver* of the software package *Mauve* by Darling et al. [Dar+04], *Mutagen* of the alignment evaluation suite *ThurGood* by Shatkay et al. [Sha+04], *Evolver* by Edgar et al. [Edg+09], and the tool *PEGsim* proposed by Yang and Setubal [YS11]. These models consider mutations such as simple indels (insertions or deletions of single base pairs) as well as transfers of longer blocks, reverse complements and several other evolutionary events.

5.4 Results

We used the described tools to create real world texts and synthetic texts, which we analyzed for their statistical properties. The resulting test instances and their statistical properties are described in the following sections and summarized in Table 6 and Table 8.

5.4.1 Texts

We concatenated subsets of the individual small real world text files to form long texts. We generated synthetic texts of length 2^{30} using the Bernoulli, Fibonacci and Markov process generator. The parameters of the Markov process were estimated from real world texts (we estimated the parameters from 2^{30} characters long prefixes of the texts to reduce the memory requirements). We used preferably high values for the order g of the Markov process but so that the exponential size q-gram table still fits into main memory. This results in the following test instances:

`text-english` Concatenation of all English texts of Project Gutenberg that were recognized as using ASCII encoding (see Section 5.1.1).

`text-german` Concatenation of all German texts of Project Gutenberg that use a Latin-1/ ISO-8859 based or ASCII encoding (see Section 5.1.1).

`text-chinese` Concatenation of all Chinese texts of Project Gutenberg, each converted to UTF-8 encoding using the Linux `iconv` command (see Section 5.1.1).

`dna-human5` Concatenation of the chromosomes of the human genome, including the IUPAC character class N (see Section 5.1.2) in the following order: Chromosomes 1 – 22, X, Y.[11]

[11] The corresponding FASTA headers are in the Appendix (Section A.2.6). First we used a different ordering in which Y and X chromosome were at the beginning. In the experimental evaluation we noted that several algorithms performed much worse on real DNA sequences than reported by the authors of the original implementations. This was caused by long repeated regions shared between chromosomes X and Y which negatively affect the performance of the algorithms (discussed below in Section 6.3.3). We therefore use the indicated ordering, so that experiments using prefixes of the sequence only work on the *autosomes* (chromosomes 1 – 22).

File name	Encoding	File size	Sequence length n	Alphabet size σ	Inv. prob. matching[10]	Alphabet Σ	Character type
`text-english`	ASCII	8520 MiB	8 934 380 282	185	15.74	“_ !"#$%&’()*+,-./0123456789:;<=>?@A. . .”	`char`
`text-german`	ISO-8859	204 MiB	214 072 799	190	14.52	“_ !"#$%&’()*+,-./0123456789:;<=>?@A. . .”	`char`
`text-chinese`	UTF-8	147 MiB	52 812 589	16 565	104.12	“一丁七万丈三上下丌不与丏丐丑专且丕世丘丙业丛. . .”	`wchar_t`
`dna-human5`	ASCII	2952 MiB	3 095 677 412	5	4.41	“ACGNT”	`Dna5`
`dna-human4`	ASCII	2728 MiB	2 861 327 131	4	3.87	“ACGT”	`Dna`
`protein-all`	ASCII	5195 MiB	5 448 131 689	26	16.32	“ABCDEFGHIJKLMNOPQRSTUVWXYZ”	`AminoAcid`
`uniform-ascii`	ASCII	1024 MiB	1 073 741 824	94	94.00	“ !"#%&’()*+,-./0123456789:;<=>?@A. . .”	`char`
`uniform-dna4`	ASCII	1024 MiB	1 073 741 824	4	4.00	“ACGT”	`Dna`
`uniform-protein`	ASCII	1024 MiB	1 073 741 824	26	26.00	“ABCDEFGHIJKLMNOPQRSTUVWXYZ”	`AminoAcid`
`uniform-binary`	ASCII	1024 MiB	1 073 741 824	2	2.00	“01”	`bool`
`fibonacci`	ASCII	1024 MiB	1 073 741 824	2	1.89	“01”	`bool`
`markov-english`	ASCII	1024 MiB	1 073 741 824	139	15.55	“ !"#$%&’()*+,-./0123456789:;<=>?@A. . .”	`char`
`markov-dna4`	ASCII	1024 MiB	1 073 741 824	4	3.89	“ACGT”	`Dna`
`markov-protein`	ASCII	1024 MiB	1 073 741 824	26	16.02	“ABCDEFGHIJKLMNOPQRSTUVWXYZ”	`AminoAcid`

Table 6: Test instances: basic statistical properties.

We composed several real world test instances and created synthetic texts using stochastic processes. The generated texts were analyzed for their statistical properties. This tables summarizes the basic properties and the results are discussed in Section 5.4.2.1 ff.

[10] Inverse probability of matching

Sequence length in characters	File size
$65\,536 = 2^{16}$	64 KiB
$262\,144 = 2^{18}$	256 KiB
$1\,048\,576 = 2^{20}$	1 MiB
$4\,194\,304 = 2^{22}$	4 MiB
$16\,777\,216 = 2^{24}$	16 MiB
$67\,108\,864 = 2^{26}$	64 MiB
$268\,435\,456 = 2^{28}$	256 MiB
$1\,073\,741\,824 = 2^{30}$	1 GiB

Table 7: Test instances: prefix lengths.

We extracted prefixes of the texts to test the pattern matching algorithms with different text sizes. The prefix lengths are powers of 2. These values correspond to the indicated file sizes if ASCII encoding (or another encoding with 1 byte per character) is used.

`dna-human4` Concatenation of the chromosome sequences of the human genome, without IUPAC character classes in the following order: Chromosomes 1 – 22, X, Y (see Section 5.1.2).

`protein-all` Concatenation of all protein sequences downloaded from the NCBI GenBank (see Section 5.1.3).

`uniform-ascii` Text generated by a Bernoulli process using all 95 printable ASCII characters, except for the $ symbol (see Section 5.3.1).

`uniform-dna4` Text generated by a Bernoulli process using the DNA alphabet of size 4 (see Section 5.3.1).

`uniform-protein` Text generated by a Bernoulli process using the extended amino acids alphabets of size 26 (see Section 5.3.1).

`uniform-binary` Text generated by a Bernoulli process using the binary alphabet $\Sigma = \{0, 1\}$, stored as ASCII text (see Section 5.3.1).

`fibonacci` The Fibonacci string generated over the binary alphabet $\Sigma = \{0, 1\}$, stored as ASCII text (see Section 5.3.2).

`markov-english` Text generated by a Markov process of order 3; the parameters were estimated from `text-english` (see Section 5.3.3 and Section A.2.5).

`markov-dna4` Text generated by a Markov process of order 9; the parameters were estimated from `dna-human4` (see Section 5.3.3).

`markov-protein` Text generated by a Markov process of order 4; the parameters were estimated from `protein-all` (see Section 5.3.3).

We additionally created prefixes of the generated texts to test the algorithms with different text lengths. This also enables us to perform experiments with sequences of exactly the same length but with different characteristics (e. g., with different alphabet sizes). The prefix lengths (counted in characters) are powers of 2 as indicated in Table 7. The resulting prefix files are named by appending the exponent to the original file name, e. g., `text-english-16`, `text-english-18`, and so on up to `text-english-32`.

5.4.2 Statistical properties

We used our text analysis tools to determine the statistical properties of all test instances. Basic statistical properties such as the file size, the sequence length, and the alphabet are given in Table 6. Statistics regarding the *q*-gram distribution, the empirical entropy, and the compressibility are in Table 8 (here we used prefixes of length 2^{30} for the longer texts so that all texts – except

File name	*q*-gram distribution			Entropy						Relative compressed size		
	1-grams	2-grams	3-grams	H_0	H_1	H_2	H_3	H_4	H_5	`gzip`	`bzip2`	`xz`
`text-english-30`	147	7 424	113 678	4.53	3.63	2.96	2.44	2.10	1.89	37.77 %	28.10 %	**23.54 %**
`text-german`	190	8 914	101 319	4.57	3.59	2.94	2.47	2.12	1.88	37.24 %	28.13 %	**21.06 %**
`text-chinese`	16 565	2 699 021	16 683 667	9.46	7.37	4.85			[12]	44.51 %	33.90 %	**30.65 %**
`dna-human5-30`	5	25	120	2.29	1.66	1.65	1.64	1.64	1.63	22.90 %	21.84 %	**18.79 %**
`dna-human4-30`	4	16	64	1.98	1.94	1.93	1.92	1.91	1.90	26.74 %	25.56 %	**21.98 %**
`protein-all-30`	26	617	10 303	4.14	4.13	4.12	4.11	4.04	3.75	55.01 %	51.42 %	**34.75 %**
`uniform-ascii`	94	8 836	830 584	6.55	6.55	6.55	*6.55*	*6.55*	*6.55*	83.01 %	**82.81 %**	83.49 %
`uniform-dna4`	4	16	64	2.00	2.00	2.00	2.00	2.00	2.00	28.56 %	27.34 %	**26.56 %**
`uniform-protein`	26	676	17 576	4.70	4.70	4.70	4.70	4.69	*4.69*	63.53 %	**59.79 %**	61.70 %
`uniform-binary`	2	4	8	1.00	1.00	1.00	1.00	1.00	1.00	14.91 %	16.01 %	**13.41 %**
`fibonacci`	2	3	4	0.96	0.59	0.37	0.37	0.23	0.23	0.44 %	**0.01 %**	0.18 %
`markov-english`	139	8 624	207 137	4.62	3.72	3.00	2.41	2.40	2.38	47.43 %	38.54 %	**38.34 %**
`markov-dna4`	4	16	64	1.98	1.94	1.93	1.92	1.91	1.90	27.37 %	26.11 %	**24.51 %**
`markov-protein`	26	625	10 906	4.14	4.13	4.12	4.11	4.04	*4.00*	58.47 %	**54.02 %**	54.42 %

Table 8: Test instances: extended statistical properties.

The test instances were analyzed for the *q*-gram distribution (counting the number of different substrings), the empirical entropy (a measure for the information content of a symbol) and the compressibility (using three compressors). The results are discussed in Section 5.4.2.4 ff.

[12] Some higher order entropies could not be computed because of the exploding memory requirements of the big alphabet; the values in *italics* are estimates.

for the shorter `text-german` and `text-chinese` – have the same length to allow an easier length-independent comparison). The presented tables are a direct output of our tools except for some formatting. Both tables are discussed in the following sections in more details.

5.4.2.1 File size

For nearly all texts, the file size in byte equals the sequence length in characters because the files have single-byte encodings and do not use the FASTA file format but a plain concatenation without headers. Only `text-chinese` uses UTF-8 encoding and the file size is therefore greater than the sequence length (one character is stored in 2.93 B = 23.40 bit on average).

5.4.2.2 Alphabet

The texts have very different alphabet sizes ranging from 2 to 16.565. Even though the test instance `text-english` consists of a concatenation of texts that were recognized as ASCII text files, it contains 185 different symbols including also non-ASCII and non-printable characters; the same holds for `text-german`.
The sequence `dna-human5` contains all four bases plus the wild-card character N (actually, it additionally also contained exactly 2 occurrences of the character class M and one occurrence of R, which we simply converted to the superclass N for harmonization).

5.4.2.3 Inverse probability of matching

The relation if the alphabet size to the inverse probability of matching is shown graphically in Figure 17. The exact values are given in Table 6 on page 119.
For all three natural language texts, the inverse probability of matching is considerably lower than the alphabet size because natural language texts tend to use a subset of characters disproportionately often and others rarely.
For `dna-human4`, the inverse probability of matching is close to the alphabet size, indicating a relatively uniform distribution of the four bases.
For the uniformly generated synthetic texts, the inverse probability of matching equals the alphabet size as expected.
For the Fibonacci string, the inverse probability of matching is less than 2 because it contains significantly more 0 than 1 symbols (the factor is the golden ratio φ [Ber86]).
For the synthetic sequences generated by a Markov process, the inverse probability of matching resembles the value of the original base sequence from which the character distribution was estimated with only small deviations.

5.4.2.4 *q*-gram distribution

The *q*-gram distributions of the test instances are shown in Table 8 on page 121. We found that for natural language texts, the number of different *q*-grams cannot easily be described by a simple function of *q* and alphabet size σ or the inverse probability of matching. For English, German, and especially Chinese we observe that only a small proportion of possible 2- or 3-grams actually occurs in the texts because not all combinations of symbols (representing *characters* in English and German and *words* in Chinese) are meaningful in the grammar of the languages.
In the sequence of the human genome, all 2- and 3-grams actually exists, whereas in the protein sequence, some combinations of amino acids do not occur at all. (In this work, we do not analyze the distribution of specific symbols, as the “CpG” content which is of mainly biological interest.)
We found that in the Fibonacci string, several *q*-grams never occur, such as “11” and “000” (which can also be deducted from the construction rules of the Fibonacci string).

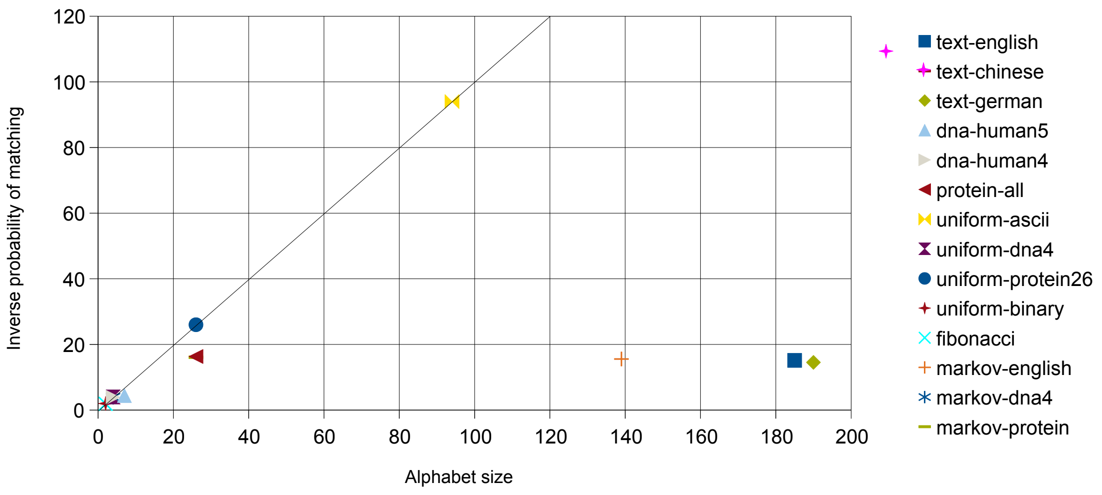


Figure 17: Test instances: alphabet size and inverse probability of matching.

The test instances have very different alphabet sizes ranging from 2 for binary to 16565 for Chinese texts (shown outside of the actual diagram).

The inverse probability of matching is a measure for the actually used alphabet in the texts and dots near the main diagonal therefore represents texts where all characters are frequently used. Especially natural language texts exhibit an inverse probability of matching which is significantly lower than the complete alphabet.

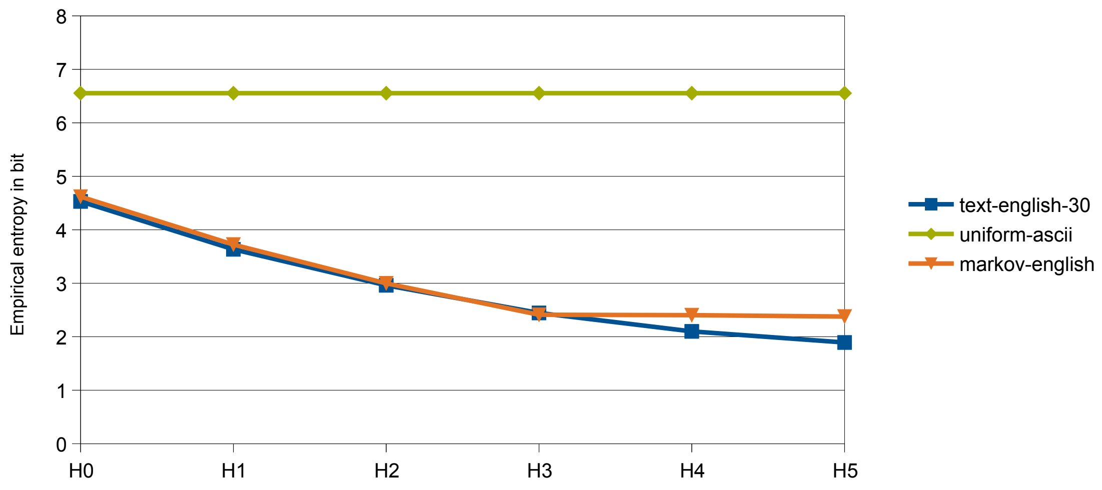


Figure 18: Test instances: empirical entropy of natural language and synthetic texts.

The empirical entropy H_g of the English text shows correlations between characters and the immediately preceding g characters. A character has, for example, less than half the information content if the preceding 4 characters are known (2.10 bit compared to 4.53 bit).

The text generated by a Markov process of order 3 trained on the English text exhibits a very similar behavior of the empirical entropy up to order $g = 3$.

The uniformly generated text has a constant empirical entropy independent on the order g because there are no (or only coincidental) correlations.

(The maximal achievable entropy with 1 B is 8 bit.)

The number of different q-grams grows for the uniformly generated synthetic texts exactly with σ^q up to $q = 3$; this is, however, only the case because the texts are sufficiently long so that each q-gram actually occurs. For higher values of q, some possible q-grams are not contained.
We analyzed q-gram distribution of sequences generated by a trained Markov process of order g in comparison with the original sequence used for training (for different values of q). We observed a very similar q-gram distribution for $q = g + 1$ [DK11a; DK11b]. This is no coincidence since the trained Markov process is irreducible and therefore the relative frequencies in a generated text will be roughly equal to the stationary distribution. Since the stationary distribution of the trained process is approximately the relative frequency distribution in the original sequence, the q-gram distributions will be similar. Also the q-gram distribution for smaller $q \leq g$ is relatively similar to the original sequence for `markov-dna4` and `markov-protein` which were generated by Markov processes of order 9 and 4, respectively. For higher $q > g + 1$ we found that the q-gram distribution is not simulated well by a Markov process of order g because only short-range correlations are modeled in this stochastic process. This can also be seen in an example of a Markov-generated text in the Appendix (Section A.2.5).

5.4.2.5 Entropy

The test instances exhibit different entropies indicating that the information density is different between, e. g., natural language texts, biological sequences and uniformly generated sequences (Table 8). We measured the entropies in two ways (as *empirical entropy* and as the *conditional entropy of the generating stochastic process*) and both yield the same results.
The natural language texts `text-english` and `text-german` show a similar behavior for different orders g (shown for `text-english` in Figure 18). We additionally analyzed texts of other Latin-based languages (French, Spanish and Latin) and observed differences which are, however, only small. We computed the entropy also for each individual book of the Project Gutenberg corpus and observed great differences *within* the languages, which result, e. g., from the fact that some books contain a lot of formatting using the space character “ ”. The entropy of `text-chinese` is significantly higher, especially due to the big underlying alphabet.[13] It is, however, not as high as one could expect (if the 16 565 characters were distributed uniformly, the entropy would be $\log 16\,565 \approx 14.0$ while it actually is 9.46); the Chinese language contains regularities regarding which symbol ($\approx$ word) follows another symbol and not all symbols are used often as indicated by the inverse probability of matching. In all languages, the entropy decreases with growing g since if the preceding g characters are known, a character is often nearly determined or is chosen only from a small set of characters (Figure 18).
For the sequence of the human genome we note that H_0 is close to 2 which would correspond to a uniform distribution of the four bases. The entropy decreases slightly for small orders and more significantly for higher orders, indicating that a base is not already determined by only a few preceding bases. Within coding regions or within repetitions a different entropy could be observed due to correlations, e. g., based on the codon structure but here we analyze the whole genomic sequence of the human genome which probably nearly cancels out such local effects.[14]
We generated diagrams similar to the ones by [Her+94; HD06] for a prefix of the human genome to analyze the entropy for higher orders (here we use a prefix of length 2^{24} = 16 MiB and observed a similar behavior for longer prefixes). It is possible to observe an ‘‘increasing decline’’ (negative second derivative) of the entropy for higher orders (Figure 19).
For the protein sequence, the entropy decreases only slowly with growing order g (not shown

[13]The entropy is measured in terms of character symbols and not in terms of bytes of the UTF-8 encoding.
[14]We focus on the entropy of `dna-human4` and not on `dna-human5` since analyzing the entropy of a sequence with wild-cards seems less meaningful.

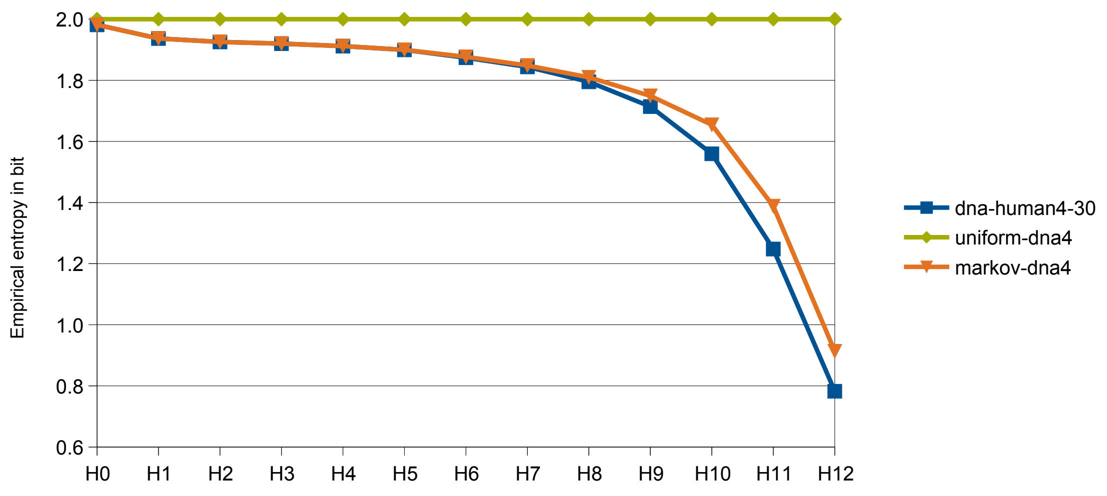


Figure 19: Test instances: empirical entropy of DNA sequences.

The empirical entropy H_g of the DNA sequence of the human genome was calculated for higher orders and shows the increasing correlations between consecutive bases. A base only has an information content of 0.78 bit if the preceding 12 bases are known (compared to nearly 1.98 bit for order 0).
The sequence generated by a Markov process of order 9 trained on a prefix of the human genome exhibits a similar behavior of the empirical entropy up to order $g \approx 8$.
The uniformly generated DNA sequence shows a constant empirical entropy of 2 bit, which is as expected the maximum achievable with an alphabet of size 2.

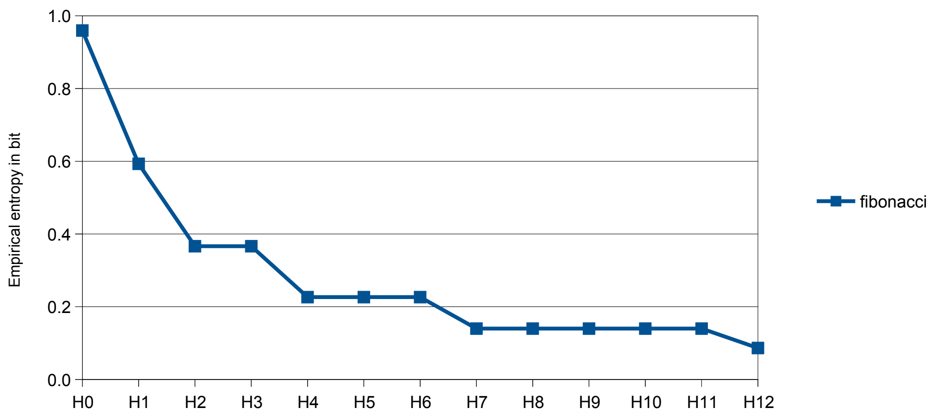


Figure 20: Test instances: empirical entropy of the Fibonacci string.

The empirical entropy H_g of the Fibonacci string was calculated for higher orders and shows a fast decline in the character correlations. It furthermore exhibits an interesting pattern because the empirical entropy H_g drops significantly compared to H_{g-1} if $g + 1$ is a Fibonacci number, and stays constant otherwise.

graphically), indicating that the choice for the next amino acid in a sequence is (statistically) not determined by the preceding few amino acids.
The uniformly generated sequences all have entropies so that $2^{H_g} \approx \sigma$. This holds independently of the order g since there are no correlations between the symbols (except for some coincidentally introduced correlations, which is why H_4 of `uniform-protein` is slightly lower).
The Fibonacci string over a binary alphabet has a 0-order entropy $H_0 < 1$ because the characters are not uniformly distributed (as discussed above for the inverse probability of matching). For higher orders g the entropy declines fast due to the repetitive structure. Interestingly we observed $H_2 \approx H_3$ and $H_4 \approx H_5$; further investigation yielded that the entropy H_g compared to H_{g-1} significantly drops if $g + 1$ is a Fibonacci number and otherwise stays on the same level. This can nicely be seen in Figure 20 where we calculated the entropy also for higher orders.
The sequences generated by a Markov process of order g show a very similar behavior of the entropy compared to the original sequences up to order $g + 1$ because the q-gram distributions are similar as discussed above. For higher orders the correlations are not modeled and the entropy of the synthetic sequences remains more or less the same, while the entropy of the original sequences tends to 0.

In general, we found that the entropy is very similar for a text and its prefixes – this holds for the synthetic texts but also for the natural language texts and biological sequences.
One general rule we observed for most texts is that there is a close connection between the entropy H_0 and the inverse probability of matching: 2^{H_0} equals approximately the inverse probability of matching. This holds for all analyzed text except for `text-chinese` where the inverse probability of matching is lower.

5.4.2.6 Compressibility

All text are compressible with each of the three compressors. For the analysis here we used the highest compression parameter `-9` and all three compressors behave rather similar on the test instances. The compressor `xz` produces the smallest result in 10 of 14 cases, `bzip2` in the other 4 cases (only synthetic texts) and `gzip` was never the best choice.[15] Here we use the relative compressed size as a length-independent measure of the compressibility. The compressibility of the test instances is graphically presented in Figure 21, a relation of the compressibility to the empirical entropy in Figure 22.
All natural language texts were compressible with a factor of 3 to 5 compared to the original size because they exhibit many regularities and repeated substrings (frequent words, idioms etc.), as well as a skewed character distribution.
For the DNA sequences we observed that interestingly the DNA sequence with larger alphabet `dna-human5-30` is better compressible, and it turned out that this is because of very long stretches of consecutive "N" characters. Since in `dna-human4-30` one byte stores one of the four bases, a compression of 25 % can already be expected only due to the small alphabet size. However, a better compression was only achieved by `xz`.
The uniformly generated text `uniform-ascii` and `uniform-protein` with large alphabets are not very compressible due to the high amount of randomness they contain. The uniformly generated binary and DNA sequences can be compressed because of the smaller alphabet.
The Fibonacci string turned out to be extremely compressible, `bzip2` even compressed it from 1 GiB to 59 604 B which is a reduction to 0.0056 % of the original size. This is possible because the Fibonacci string is by construction highly repetitive which can be exploited by the compressor.

[15] However, also the running times of the algorithms were different but here we are only interested in the compressibility of the texts.

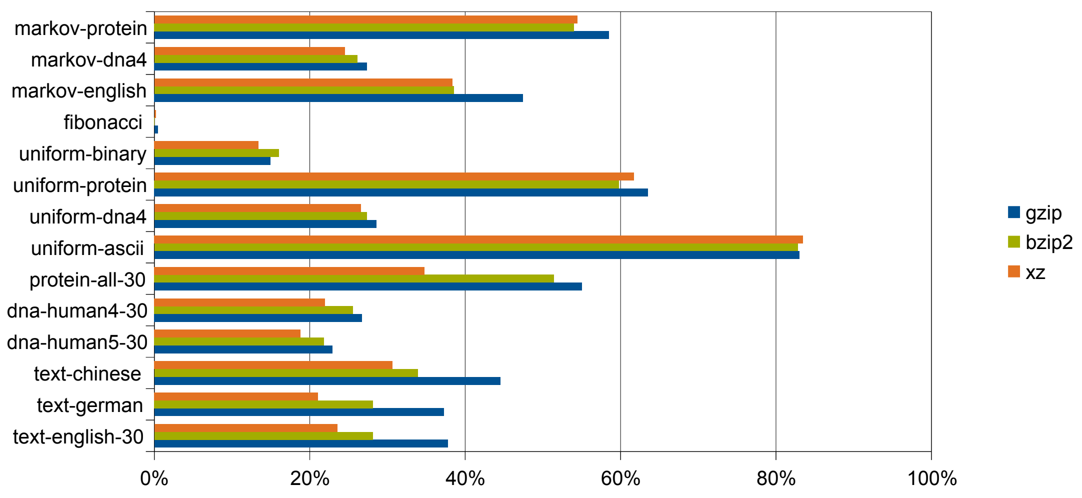


Figure 21: Test instances: relative compressed size with different compressors.

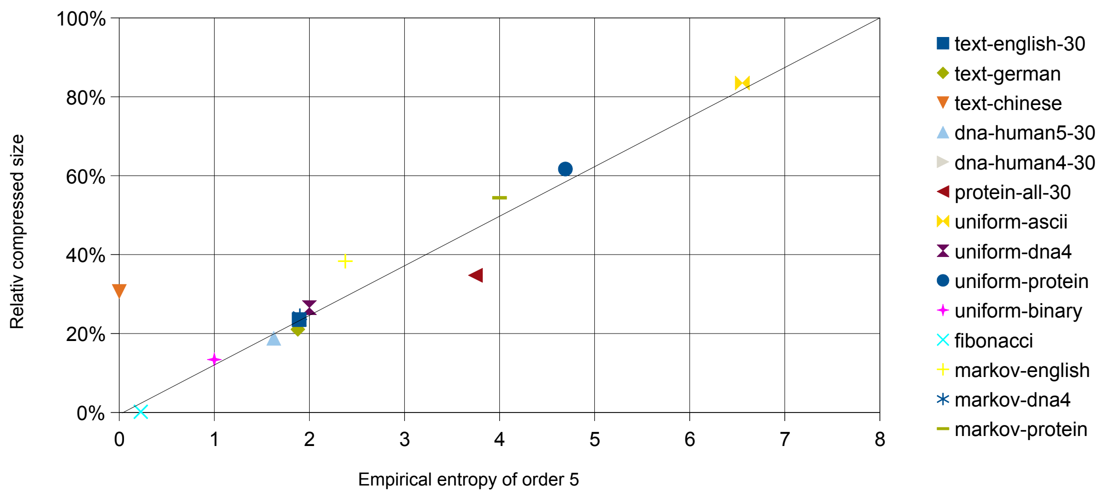


Figure 22: Test instances: relative compressed size in relation to the entropy.

A close connection of the compressed size (using for each file the respective best compressor) to the entropy (here exemplary the empirical entropy H_g of order $g = 5$) can be observed: the higher the entropy, the bigger the compressed file. The diagonal represents the lower bound given by the empirical entropy that can be achieved by any compressor considering only a context of length g preceding each character. Some compressed sizes are below this lower bound because the compressors are not limited to a context of length 5.
(The lower bounds does not apply for `text-chinese` since it uses a multi-byte UTF-8 encoding and the entropy is calculated for characters and the file size is measured in bytes.)

5.5 Pattern generator

A complete test instance for approximate pattern matching consists not only of a text but also requires a set of search queries. A search query consists of the pattern string, the similarity measure, and a search tolerance (see Section 1.4.2). We designed and implemented a search pattern generator that outputs search patterns for a given input text. The pattern generator takes as input a text file and parameters defining the number of patterns and their length m, as well as a similarity/distance measure δ and a search tolerance k. The patterns are generated by extracting text substrings of the specified length where the starting positions in the text are chosen randomly. This process already yields a set of search patterns.
However, when using the patterns for *approximate* pattern matching, it is not desirable that all patterns occur unchanged in the text (as an exact match) because this is not a realistic scenario. All boolean queries would in this case, for example, yield true as result. We therefore apply some perturbations to the extracted substrings, i. e., we transform the substring to an element of the k-neighborhood of the string (see Section 3.3.1 on page 83). We implemented this perturbation for the Hamming distance (Section 3.1.1) and the simple edit distance (Section 3.1.2):

- Hamming distance: The substring of the text is altered by applying k substitutions of single characters. A new character is selected randomly and uniformly. (When substituting a character u with w we do not explicitly exclude the case $u = w$.)
- Edit distance: The substring is altered by applying k operations where an operation is the insertion, deletion or substitution of a single character.

The alphabet for new characters (in the operations *insert* and *substitute*) is the actually used alphabet of the text (which is collected in an initial linear scan) so that the resulting patterns are therefore a set of strings over the actually used alphabet of the text.

(The resulting patterns are not necessarily of length m when using insertions and deletions. To achieve that all resulting patterns are of the same length m we extract substrings of length $m + k$ and output the prefix of length m of the modified strings.)
The pattern generator works with plain text files and FASTA formatted files and can also be used with UTF-8 encoded files. The generated patterns are written in a file and are separated by \$ symbols (we therefore do not extract patterns that already contain this symbol).

Pattern sets. For each test instance (Table 6) we used our pattern generator to create pattern sets (we used the prefix of length 1 GiB for larger files[16]). The sets have different characteristics and contain 1000 patterns each. We try to cover most relevant practical scenarios of approximate pattern matching (described in the motivation of this thesis, Section 1.1) and therefore use a broad range of values for the parameters:

- Pattern length $m \in \{4, 16, 64, 256, 1024\}$
- Distance measure $\delta \in \{\delta_{\text{Hamming}}, \delta_{\text{edit}}\}$
- Tolerance $k \in \{0, 1, 2, 3, 4, 8, 16, 32, 64, 128, 256\}$
- Error level $0 \leq \alpha \leq 25\,\%$, where $\alpha := \frac{k}{m}$

We use the restriction regarding the error level because for high error levels, very many (and ultimately all) text positions match a pattern [Wan+14]. For high error levels *online* algorithm are faster than *offline* [Nav+01]. In this thesis we are, however, mainly interested in pattern matching *with index structures* and therefore restrict the error level.

[16]For `dna-human4` and `dna-human5` we first used a different ordering of the chromosomes (as described in Section 5.4.1) and created the pattern set for this instance.

Pattern length m	Used tolerances k										
4	0	1									
16	0	1	2	3	4						
64	0	1	2	3	4	8	16				
256	0	1	2	3	4	8	16	32	64		
1024	0	1	2	3	4	8	16	32	64	128	256

Table 9: Search patterns: combinations of pattern lengths and tolerances.

This yields 34 combinations of pattern length and tolerance for each of the two distance measures (Table 9). The total number of pattern sets for each text is therefore 68.

The file names of the pattern sets are composed as follows: `patterns_`t`_`m`_`δ`_`k`.txt`, where t represents here the name of the underlying text file, yielding, e. g., the following file name: `patterns_text-english_64_edit_4.txt`. An example pattern file can be found in the Appendix (Section A.2.7).

Alternatives. There are many possible alternatives or extensions to our pattern generator. For natural language texts, one could, for example, use subsets of spelling dictionaries. To introduce even more realistic spelling mistakes in the patterns, it is possible to use a compilation of very frequent spelling mistakes made by human (in scenarios where this is important). A set of 170 human-generated spelling errors was compiled, e. g., by Kukich [Kuk92] (as cited by [ZD95]).
A very common application of approximate pattern matching in DNA sequences is *fragment assembly* during genome sequencing. The so-called *reads* of the biochemical process are used as patterns and have very specific properties based of the respective sequencing method. The most outstanding property is the read length but also the distribution of errors varies and the error probability can, e. g., increase for higher positions or in consecutive runs of the same base. Archives of reads and other short DNA sequences can be found among others in the NCBI GenBank (Expressed Sequence Tag database, Short Read Archive, . . .) [Nat09]. Tools to synthetically generate reads from given DNA sequences, e. g., for testing the quality of sequence assembly algorithms are, for example: *celsim* by Myers [Mye99a], *FASIM* by Hur et al. [Hur+06], *MetaSim* by Richter et al. [Ric+08], and *Mason* by Holtgrewe [Hol10].

5.6 Summary

We provide real world test instances of different types and characteristics, where we harmonized the file format and alphabets, removed additional header information, selected interesting subsets and concatenated the individual texts to form one long text file each. We additionally created prefixes of different lengths and analyzed all texts for several statistical properties. The texts and their statistical properties are summarized in Table 6 on page 119 and Table 8 on page 121. Our pattern generator allows to provide search patterns for each text; the sets of search patterns have different characteristics and are summarized in Table 9.
The combination of our text analysis and text generation tool allows to generate sequences which are similar to real world sequences with respect to certain properties. This allows to investigate the performance of pattern matching algorithms under to some extent realistic, yet controlled conditions, and to determine the degree of dependence from parameters of the underlying sequence. Both tools have an extensible design which allows the integration of new modules for other statistical properties or generating models with the same programming interface.

6 Experimental evaluation

The experimental evaluation brings the index structures and approximate search algorithms together with the test instances. After a description of the benchmarking environment, we examine the practical performance of each index structure and search algorithm in different settings. Then we change the perspective and determine for representative settings the respective best index structure and search algorithm.

6.1 Benchmarking framework

To efficiently perform benchmarks with the index structures and algorithms we implemented a framework consisting of a *benchmarking program* and a *benchmarking script*. One run of the benchmarking program consists of the following four general steps:

1. Read text file
2. Construct index (or do nothing when benchmarking online algorithms)
3. Read pattern file
4. For each pattern: run approximate pattern matching algorithm (in our benchmark we run 1000 different queries)

The benchmarking program has the following parameters:

- Step 1
  - Text file
  - Alphabet type:[1] `char, wchar_t, Dna, AminoAcid, bool`
- Step 2
  - Name of the index structure
  - Parameters of the index and the construction algorithm (where applicable)
- Step 3
  - Pattern file
  - Distance measure: `edit, hamming`
  - Search tolerance k
  - Maximal error level α
- Step 4
  - Query type: `bool, count, pos`
  - Name of the approximate search algorithm
  - Parameters of the approximate search algorithm (where applicable)

It is possible to specify multiple values for the parameters of Step 3 and Step 4, so that both steps are executed repeatedly for all combinations without a need for re-construction.
The *benchmarking script* is used to automatically compile and run the benchmarking program; this makes it possible to efficiently test several index parameters since many parameters need to be defined at compile time due to the template programming used in the software library. (Each compilation of the program takes on average for the different index structures approximately 1–2 min.)

[1] In our experiments we always use the best fitting alphabet type, see Table 6 on page 119.

We measure the construction time, space usage, construction space, search time, and search memory for each index structure X and pattern matching algorithm Y as follows:

$$\begin{aligned}
T_X^{\text{construct}} &= (\text{time needed for Step 2})\\
S_X &= (\text{memory needed after Step 2}) - (\text{memory needed before Step 2})\\
S_X^{\text{construct}} &= (\text{maximal memory needed during Step 2}) - (\text{memory needed before Step 2})\\
T_Y^{\text{bool/count/pos}} &= (\text{time needed for Step 4})\\
S_Y &= (\text{maximal memory needed during Step 4}) - (\text{memory needed before Step 4})
\end{aligned}$$

The times were measured as wall clock time, system time, and user time; all values given in the following refer to the *wall clock time* since this is the most useful measure when comparing algorithms for their practical performance. All indicated measured times are the arithmetic mean of at least 3 measurements / search queries.
One problem we encountered during the tests was that the first (or the first few) executions of approximate search algorithms were systematically slower than succeeding executions with the same index. This turned out to be based on caching effects. Before each approximate search (Step 4), we therefore clear the Level 2 cache of the CPU[2] and for external memory algorithms also the hard disk buffers[3] as well as all buffers implemented in the software library SeqAn[4]. This way, each approximate search starts under the same conditions with empty caches.

The space is measured with two different approaches, one from the outside and one from the inside of the benchmarking program (both methods were already available in the software library): The first approach uses operating system information (on Linux it uses `/proc/self/status`) to measure the current and maximal virtual memory usage of the program. The second approach records calls to memory allocations and deallocations and therefore measures the heap memory used; the memory of the stack is not counted. All values given in the following refer to the first approach since this value is the actual limiting factor regarding the memory usage. (We do not count space used in external memory since in most practical settings this is not the limiting factor.)
For measuring the space usage S_X of an index structure we decided to explicitly free unused space and temporary data structures using the *shrink-to-fit* and *clear-and-minimize* idioms to get preferably expressive values [Wik15].

All data of the benchmarks is logged in CSV files (comma-separated values). This includes the date and time of the benchmark and all parameters, as well as the measured time and space usage.

6.2 Benchmarking environment

The benchmarks were run on a regular desktop computer with the following setup:

CPU: AMD Athlon$^{\text{TM}}$ XP 3000+ with one core at 2154 MHz (Level 1 cache: 64 KiB, Level 2 cache: 500 KiB)

Main memory (RAM): 1475 MiB with a memory page size of 4 KiB

Operating system: openSUSE 12.2 (Mantis)

[2]Loading a sufficiently big array into main memory.
[3]Using the Linux command `drop_caches`.
[4]Using the function `flushAndFree()`.

Compiler: g++ (GCC) version 4.9.0 with the following flags (among others): `-O3 -Wall -pedantic` (performing best possible optimizations and displaying all warnings to comply with the ISO C++ standard)

SeqAn: revision number 14828 (2014-08-27), with debugging features disabled and profiling functionality enabled

For reading UTF-8 files we use functions of the Boost C++ libraries [Boo14] (on our computer in version 1.49).
For development and for testing cross-platform compatibility we additionally used a computer running Microsoft Windows XP and MinGW together with g++ (GCC) version 4.6.2.
For profiling of the algorithms we used the tools Very Sleepy (on Windows) and gprof (on Linux).

6.3 Index structures

We examine for each implemented index structure its practical performance. For each index structure we therefore try different parameter settings and measure the construction time, construction space and index space for all test instances. The results are analyzed depending on the properties of the underlying text, among others the alphabet type, the alphabet size, the text entropy, its compressibility, and the number of different q-grams.
Subsequently the behavior for longer texts is examined; this includes the dependence on the text length, the behavior in external memory and the maximal possible text length on our computer.
Based on the results we give recommendations for the best (or at least good) parameters settings. We try to give the recommendations so that they are not limited to our computer but more generally applicable. The further experiments then use these respective parameter settings.
Where applicable we also compare our implementation to the original implementation of the authors of an index structure.

The underlying text is held in main memory throughout all experiments since this is required by most algorithms to efficiently construct the index structures. We do not examine the case when the text is held in secondary memory. We furthermore do not use the self-index functionality of the compressed index structures which would allow to completely discard the text after construction.

6.3.1 Classical suffix array

We performed several experiments with the plain suffix array (`IndexEsa`, Section 2.1.1). We tested all construction algorithms for all test instances using texts of length $n = 2^{20} = 1$ MiB. The results are given in Table 10. It turns out that `LarssonSadakane` is the fastest algorithm for nearly all instances except for those with a small alphabet $\sigma \leq 4$ where `BwtWalk` is fastest or very competitive (here this includes the binary and DNA sequences). The algorithm `ManberMyers` is 16 to 28 times slower than the respective best algorithm for each test instance. The simple `SAQSort` is competitive, but only for small texts due to the super-linear running time (in our experiments the algorithm did not finish within one hour for a texts $n = 2^{24}$); it furthermore performs very poorly for the Fibonacci string, presumably because of the highly repetitive structure which leads to costly suffix comparisons. `Skew7` is not the best algorithm for any text, but has always a competitive running time, making it the algorithm of choice if an algorithm has to be chosen without knowing anything about the text.
Most algorithms perform faster for the uniformly generated texts (with a higher entropy) compared to their real world counterparts with skewed character distributions (with lower entropy); the opposite is, however, true for `ManberMyers` and `BwtWalk`, which are faster if the character distribution is not uniform.

This general picture is very similar for longer texts. For those experiments we chose to store the suffix array in external memory (with buffers as large as necessary and possible within the limits of main memory, using a page size of 16 MiB) and give only the results of the three most competitive construction algorithms `LarssonSadakane`, `Skew7`, and `BwtWalk`. The results are presented graphically in Figure 23. The dependency of the construction time depending on the text length is presented in Figure 24. The results indicate a linear (or slightly super-linear) dependency of the practical construction time on the text length for all three algorithms. This confirms the theoretical worst case running times (see Table 1 on page 21).
Since we want the text to reside in main memory, the algorithm can only use the remaining part of the main memory. With this setting it was possible to construct the suffix array for $n \leq 2^{26}$ using `LarssonSadakane` and `BwtWalk` and for $n \leq 2^{28}$ using `Skew7`. (If we also allow to store the *text* in external memory, it is possible to construct the suffix array for longer texts.)

The resulting index needs $4\,n$ B space for all construction algorithms. When using the main memory storage of the suffix array (`Alloc`), the construction space of the algorithms is between 4.0 and $15.0\,n$ B, see Table 11.
When using the external memory storage, the construction memory for larger texts grows only slowly, `Skew7` with `text-english-28` (of size 256 MiB) needs 870 MiB of main memory in addition to the text, which equals $3.4\ n$ MiB (this relative value shrinks with increasing text length).

Recommendation for index parameters of `IndexEsa` (suffix array):

1. `TIndexStringSpec`: Choose `Alloc` for small texts and `External` if the construction cannot be carried out in main memory (the buffer are set to contain as many frames as necessary within the limits of the main memory and we use a page size of 16 MiB).

Recommendation for construction parameters of `IndexEsa` (suffix array):

1. `TAlgSpec`: The optimal strategy for selecting an algorithm is to choose the algorithm `LarssonSadakane` for alphabets of size > 4, `BwtWalk` for smaller alphabets, and `Skew7` for longer texts (if the other algorithms fail due to the memory requirements, in our case if $n \geq 2^{26}$). We use this strategy for our further experiments with suffix arrays.

6.3.2 Suffix array with LCP table

We built the suffix array with LCP table (Section 2.1.2) for all text types and the total construction time increased by a factor between 1.5 and 2 compared to the construction time for building only the suffix array. We observe a slightly super-linear construction time for the LCP table with increasing text length. We did not observe a significant influence of the alphabet size or entropy of the text on the construction time of the LCP table.
The resulting index including the suffix array and the LCP table takes $8.0\,n$ B of space for all texts. The construction space is dominated by the construction space of the algorithm for building the suffix array, see above.

[5]The algorithm `BwtWalk` is not implemented for wide characters and would probably not perform well due to the big alphabet.

Text	SAQSort	ManberMyers	LarssonSadakane	Skew3	Skew7	BwtWalk
text-english-20	1.11	13.18	**0.77**	2.85	2.58	4.89
text-chinese-20	1.21	13.53	**0.65**	3.56	3.43	[5]
text-german-20	1.17	13.23	**0.82**	2.97	2.62	3.59
dna-human4-20	1.42	11.86	0.95	2.59	2.12	**0.61**
protein-all-20	2.51	12.23	**0.81**	2.96	2.64	1.99
uniform-ascii-20	0.90	14.27	**0.55**	2.01	1.29	9.54
uniform-dna4-20	1.29	13.40	0.93	1.94	1.87	**0.80**
uniform-protein-20	0.97	14.09	**0.66**	1.52	2.43	2.98
uniform-binary-20	1.74	12.59	1.00	1.91	1.65	**0.51**
fibonacci-20	3198.80	7.43	3.25	1.89	1.63	**0.26**
markov-english-20	1.10	13.62	**0.73**	2.64	2.32	5.50
markov-dna4-20	1.29	12.98	0.91	2.38	1.83	**0.77**
markov-protein-20	0.99	13.67	**0.67**	2.19	2.44	2.64

Table 10: Suffix array: construction time for different algorithms and texts.

(Using $n = 2^{20}$.) The time of the respective best algorithm is given in **bold**.

Text	SAQSort	ManberMyers	LarssonSadakane	Skew3	Skew7	BwtWalk
text-chinese-20	7.0	15.0	13.0	13.2	13.2	[5]
all single-byte encoded texts	4.0	14.0	12.0	12.0	8.3	8.0

Table 11: Suffix array: construction space for different algorithms and texts.

(Measured in n B.)

A comparison of the construction time of the classical suffix array, the suffix array with LCP table and the enhanced suffix arrays can be found in Figure 24.

Since the search time of the suffix array with LCP table was found by Weese [Wee12] to be not competitive to the classical suffix array and the implementation did not work together with external memory strings, we do not include it in the further experiments.

6.3.3 WOTD suffix tree

For the WOTD suffix tree (write-only top-down) (`IndexWotd`, Section 2.2.4) we test in particular the three construction variants with `eager` construction of the whole tree, construction of the `first` level only and completely `lazy` construction where only constant work is done initially.
First we describe the results for in-memory construction (using `TIndexStringSpec = Alloc`).
We constructed the index for all three variants and all test sets of length 2^{24}. The `first` construction times are very low (because the *suffixes* array is basically only traversed once) and do not differ much among the test instances (varying with a factor < 2).
The `eager` construction times depend especially on the alphabet size of the underlying text: the construction for `uniform-binary-24` takes 4 times, the construction of `uniform-dna-24` takes 2 times more than `uniform-ascii-24` (Figure 25).

The `eager` construction turned out to be very susceptible for long repeats. The eager construction time of `text-english-24` (406 s) is remarkably different compared to `text-german-24` (36 s) and `markov-english-24` (29 s) and we therefore investigated the reason of this. We finally tracked it down to a book of length about 430.000 characters which occurs twice in `text-english-24`.
The WOTD algorithm has to calculate repeatedly longest common prefixes and the time necessary for this task grows quadratically with the length of a repeated substring.
To further investigate this we carried out a series of other experiments and observed the behavior for different real-world DNA sequences. Building the WOTD suffix tree for DNA sequences containing the human chromosomes X and Y (of lengths 144 MiB and 25 MiB) was not possible within several hours. We first suspected it might be due "*repetitive elements of the LINE*[6] *type, in which the X-chromosome is particularly rich.*" [Lyo98] or because "*more than 50 % of the human Y chromosome is composed of a variety of repeated DNAs*" [Smi+87]. However, building the WOTD tree for each of the chromosomes individually was possible within a few minutes (23.6 min and 1.4 min). The significantly higher construction time therefore has to be due to an effect caused by the combination of both chromosomes. It is suspected that earlier in the evolutionary history, chromosome Y was homologue to chromosome X, and even though both chromosomes have many differences now, they still have similarities [EMG01]. Relevant here seems to be a so-called ***p**seudo**a**utosomal **r**egion* (PAR1), which is a sequence of length 2.6 Mbp and contained in both chromosomes [MM07]. This explains why building the WOTD suffix tree is infeasible for the concatenation of both chromosomes, but no problem for each individual chromosome.
The construction algorithm is obviously susceptible when having long repetitions, which is also confirmed by the long construction time for the Fibonacci string that basically only consists of repetitions (it was not possible to build the index for the Fibonacci string of length 2^{24} within one hour). An idea to make the construction algorithm work better with long repetition is therefore sketched by Giegerich et al. [Gie+03].

The space usage when building only the `first` level of the tree is 4 n B and the same for all texts because the size of the *suffixes* array depends only on the text length. When using `eager` construction, the complete tree takes between 9.5 (for `uniform-ascii`) and 17.4 n B (for

[6] **L**ong **i**nterspersed **n**uclear **e**lements are of length 6000 - 7000.

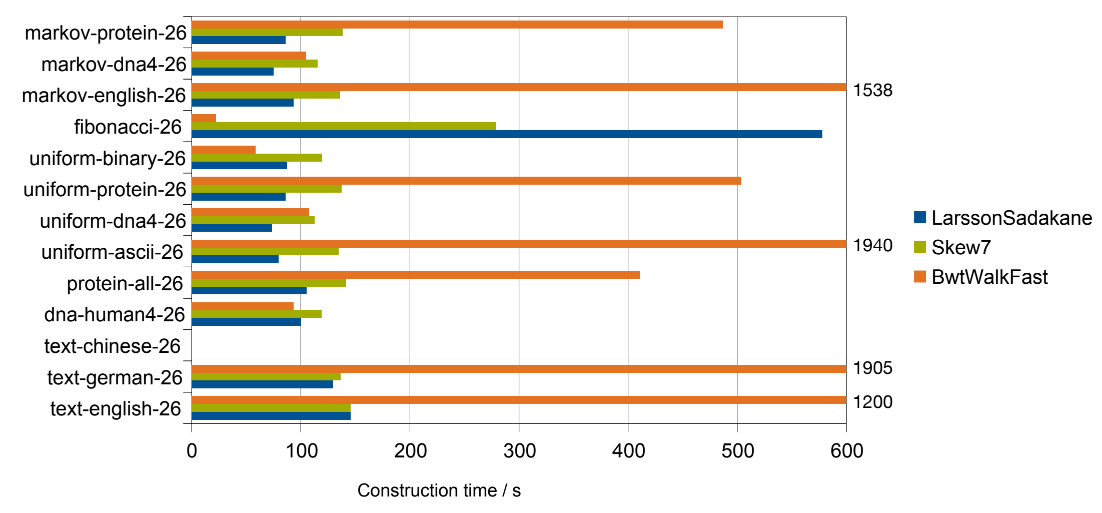


Figure 23: Suffix array: construction time for longer texts.

(Using $n = 2^{26}$.)

The time axis is cut at 10 minutes since we are interested in the *best* algorithms for each text.

The test instance `text-chinese` is shorter than 2^{26} and therefore not tested here.

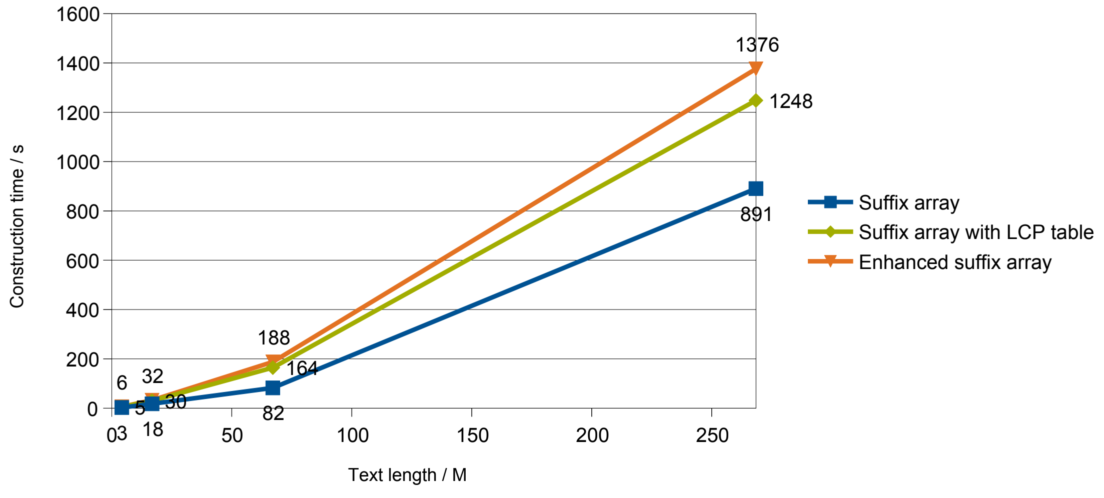


Figure 24: Suffix array and variants: construction time depending on the text length.

(Using t = `uniform-ascii` and $n \leq 2^{28}$.)

The practical construction time is roughly linear in the area where all data-structures fit into main memory, and becomes super-linear as soon as parts of the data structures need to be kept in secondary memory (especially due to the construction of the suffix array). The picture is very similar for all other types of texts.

`uniform-binary`),[7] with a construction memory between 13.3 and 29.4 n B. It is relatively smaller for larger alphabets because the tree contains less internal nodes due to the higher branching factor.

We were able to build the WOTD suffix tree in main memory with eager construction for texts up to length $n \leq 2^{26}$ and with only the first level for $n \leq 2^{28}$. The eager construction algorithms shows a slightly super-linear dependency of the construction time on the text length (Figure 27) which fits to the $\mathcal{O}(n \log n)$ expected time [Gie+03] (this holds independently of external memory effects also in the range where all data structures fit into main memory). The space grows linearly (Figure 28) with the text length.

Storing the tree (*suffixes* array plus nodes table) in external memory makes it possible to build the tree for longer texts. We use the buffering strategy proposed by Tian et al. [Tia+05] and also used by Aumann [Aum11] to distribute the available main memory frames to the data structures. The frames are distributed in the following order until no more free frames are available:

1. Text
2. Suffixes array (using a minimum of σ frames)
3. Temporary array (using a minimum of σ frames)
4. Nodes array (using a minimum of 2 frames)

This enables us to build the tree for texts where the complete *suffixes* array as well as parts of the temporary data structures and the nodes table can be held in main memory (in particular 4 n B < main memory). On our computer we were able to build the tree for texts of length 2^{28} in 967 s = 17 min, which is due to the external memory accesses already significantly slower compared to 78 s for in-memory construction of texts with length 2^{26}. We use a page size of 4 MiB and for texts of length 2^{28} the following buffer sizes:

1. Text: $65 \cdot 4$ MiB = 260 MiB (The benchmark stores the text in main memory.)
2. Suffixes array: $233 \cdot 4$ MiB = 932 MiB
3. Temporary array: (The implementation stores the temporary array in main memory.)
4. Nodes array: $2 \cdot 4$ MiB = 8 MiB

(When using the lazy version, the WOTD index can be used for considerably longer texts.)

Recommendation for index parameters of `IndexWotd`:

1. `TIndexStringSpec`: Choose `Alloc` if the complete tree fits into main memory and `External` with a configuration according to the buffering strategy of Tian et al. [Tia+05] and a page size of 4 MiB otherwise.

Recommendation for construction parameters of `IndexWotd`:

1. The variant for building the tree (`lazy`, `first`, of `eager`) should be chosen depending on the application, i. e., whether a shorter construction time and smaller index size is more important or a faster search time.

[7]At first we measured a space consumption which was significantly higher than theoretically predicted. This was caused by temporary arrays which were not completely freed after the full construction of the tree.

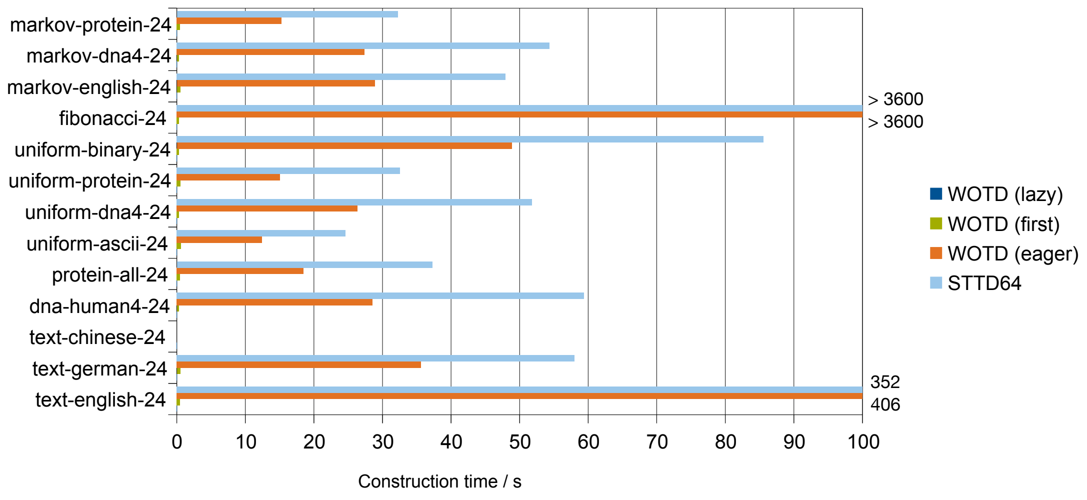


Figure 25: WOTD and STTD64 suffix tree: construction time for different texts.

(Using $n = 2^{24}$ and the following parameters for WOTD: in-memory storage using `TIndexStringSpec=Alloc`, and for STTD64: sufficiently large buffers and the in-memory variant `inMem=true`.)

The small constant time for the lazy and first level constructions are barely visible and the time axis is cut at 100 seconds.
`text-chinese-24`: The implementations do not work for wide characters.
`fibonacci-24`: The eager construction did not finish within one hour.
`text-english-24`: The eager construction time of (406 s) is remarkably higher compared to `text-german-24` (36 s) and `markov-english-24` (29 s) which is due to a repeated substring as discussed on page 136.

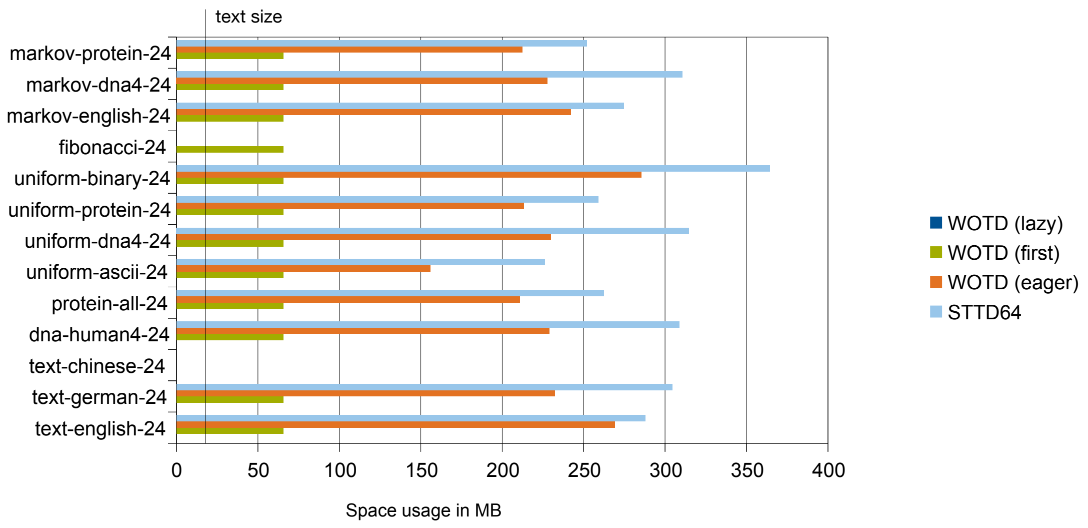


Figure 26: WOTD and STTD64 suffix tree: space usage for different texts.

(Using $n = 2^{24}$ and the following parameters for WOTD: in-memory storage using `TIndexStringSpec=Alloc`, and for STTD64: sufficiently large buffers and the in-memory variant `inMem=true`.)

6.3.4 STTD64

The index STTD64 suffix tree representation (`IndexSttd64`, Section 2.2.5) is closely related to the WOTD index discussed above. The relative construction times for the different texts of length 2^{24} are therefore similar, see Figure 25 (for such rather small texts, the construction of the STTD64 index takes about twice as long as for WOTD). Since the resulting indexes fit into main memory, we use the in-memory variant (`inMem = true`) with sufficiently large buffers (text 17 MiB, *suffixes* array 65 MiB, temporary array 65 MiB, *nodes* array 257 MiB, all with a page size of 1 MiB). The space usage of the STTD64 representation is with 13.8 n B to 22.2 n B bigger than for WOTD (about 5 n B more), see Figure 26.
The STTD64 construction algorithm has the same problem with long repeated substrings (concerning, e. g., the Fibonacci string and chromosomes X and Y), because the algorithms are very similar.

For longer texts, the complete index cannot be held in main memory, so we use the variant optimized for external memory with `inMem = false` (on our computer the turning point is for $n \geq 2^{28}$). The buffers are again distributed according to the strategy of Tian et al. [Tia+05] as proposed also by Aumann [Aum11] (e. g., for `uniform-ascii-26`: text 68 MiB, *suffixes* array 260 MiB, temporary array 260 MiB, and *nodes* array 612 MiB, all with a page size of 4 MiB). For even longer texts, it is not even possible to hold the *suffixes* array and the temporary array in main memory, so the text suffixes are partitioned depending on their prefix of length `PREFIX_LENGTH` (on our computer the turning point is as well for $n \geq 2^{28}$). This enables us to build the STTD64 suffix tree for $n = 2^{30} = 1$ GiB, even if the text is held completely in main memory and our computer only has 1.47 GiB of RAM, leaving only $\approx$ 350 MiB for the index construction after subtracting space for the operating system etc. (For, e. g., `uniform-ascii` we use `PREFIX_LENGTH = 1` and for `uniform-dna4` we use `PREFIX_LENGTH = 3`, resulting in 94 and 64 partitions of sizes 11 MiB and 16 MiB, respectively.)
We observed that if `PREFIX_LENGTH` is sufficiently high (so that the data structures for one partition fit into main memory), the construction time remains very similar even if a higher value is chosen. This is beneficial in the following scenario: If we do not know the characteristics of the text we cannot determine the size of the partitions from the prefix length (this happens, for example, if some prefixes occur more often than others); in these cases we can in doubt choose a higher value for `PREFIX_LENGTH`. In our experiments we try to always use the lowest possible value, so that the resulting data structure is as close as possible to a single suffix tree of the text.
The construction time for longer texts is presented in Figure 27. While the STTD64 index is slower than WOTD for smaller texts ($n \leq 2^{26}$), it is faster for longer texts. It can furthermore be used also for texts which are too long to be handled by WOTD with eager construction at all (e. g., $n = 2^{30}$). The space consumption depending on the text length is shown in Figure 28 (using the peak construction memory).
The obtained results for longer texts indicate that our implementation of the construction algorithm mainly done by Aumann [Aum11] is competitive to the original implementation by Halachev et al. [Hal+07]: For, e. g., the DNA sequence of the human chromosome 1 they report to need 15 min and our implementation needs 23 min. Their computer has a faster CPU (3.0 GHz vs. our 2.15 GHz) and more main memory (2.0 GiB vs. 1.4 GiB). This and the further experiments performed here and by Aumann [Aum11] indicate that our implementation is competitive.

Recommendation for index parameters of `IndexSttd64`:

1. `inMem`: `true` if the complete index fits into main memory, and `false` otherwise.

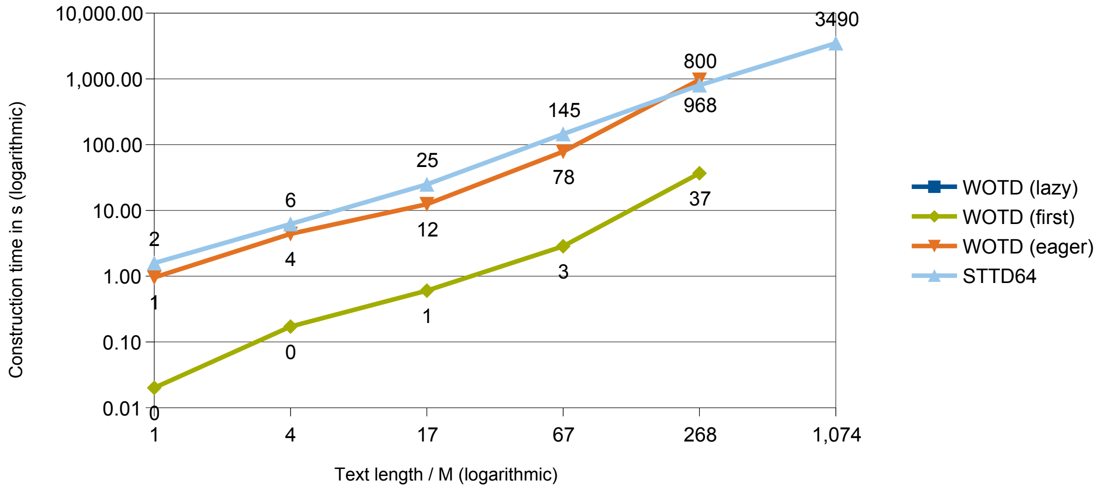


Figure 27: WOTD and STTD64 suffix tree: construction time depending on the text length.

(Using t = `uniform-ascii` and $n \leq 2^{30}$.)

The WOTD index fits into main memory for $n \leq 2^{26}$, so we use `TIndexStringSpec=Alloc`. For longer texts we use `TIndexStringSpec=External` with a page size of 4 MiB. This results in a significantly slower construction for $n = 2^{28}$. The WOTD index cannot be constructed for $n = 2^{30}$ due to memory restrictions. (The "lazy construction" takes nearly 0 time and is therefore not visible here.)
The STTD64 index fits into main memory for $n \leq 2^{26}$, so we use `inMem=true`. For longer texts we use `inMem=false` and a partitioning with prefix length 1, resulting in 94 partitions which can efficiently be handled individually in main memory.

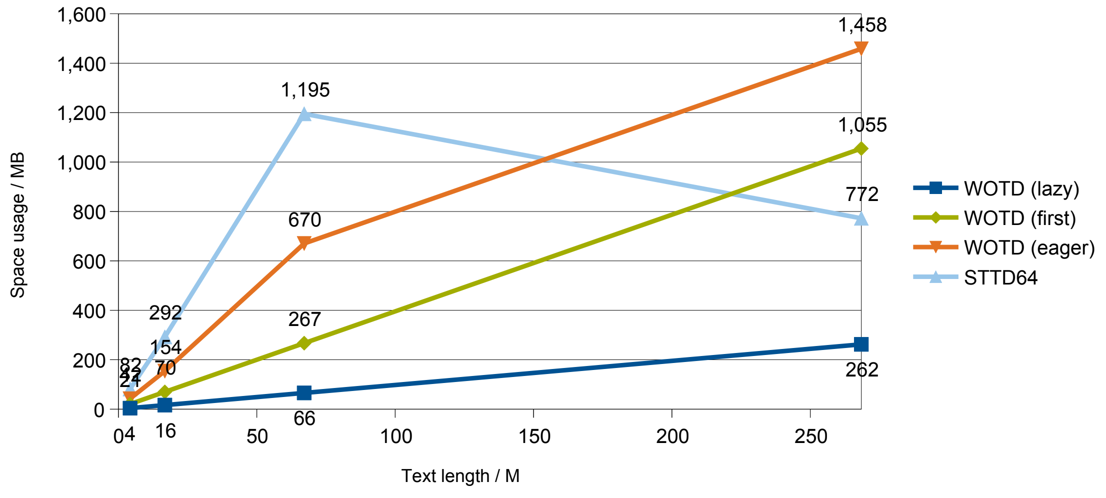


Figure 28: WOTD and STTD64 suffix tree: memory usage depending on the text length.

(Using t = `uniform-ascii`, $n \leq 2^{28}$ and the same parameters as in Figure 27.)

Here we measure the peak total virtual memory used during the construction including the text.
The complete WOTD index does not fit entirely into main memory for $n = 2^{28}$. STTD64 uses a partitioning for $n = 2^{28}$, so the peak of the used construction memory is lower than for $n = 2^{26}$ and always within the limits of the main memory.

2. PREFIX_LENGTH: Choose the lowest value so that all data structures for each of the resulting partitions fit into main memory (this includes the whole text, the *suffixes* array, and the temporary array plus 2 frames for the *nodes* array). The resulting number of partitions is $|partitions| = \sigma^{\texttt{PREFIX\_LENGTH}}$, yielding an average partition size of $n / |partitions|$. Since the partitions are not necessarily of equal size (even less for real world data with skewed character distributions), this should be accounted for with a higher value for PREFIX_LENGTH if necessary.

Recommendation for construction parameters of IndexSttd64:

1. PAGESIZE: By default we use a page size of 4 MiB.
2. TREEBUFF_FRAME_NUM,
3. SUFBUFF_FRAME_NUM, and
4. TEMPBUFF_FRAME_NUM should be assigned according to the buffering strategy of Tian et al. [Tia+05] (see above in Section 6.3.3).

6.3.5 Enhanced suffix array

The enhanced suffix array consists of the suffix array, the LCP table, and the child table (IndexEsa, Section 2.2.6). The construction time therefore is the sum of the construction times of the individual components and the same holds for the space usage (which grows linearly with the text length and is $12\,n$ B for all texts). The construction time of the child table is relatively small compared to the other components, see Figure 24 on page 137.
The enhanced suffix array fits together with the text into the main memory of our computer for texts up to length 2^{26}. The maximal possible text length is limited by the construction of the suffix array (see Section 6.3.1).

6.3.6 Suffix forest in external memory

The suffix forest in external memory (IndexDigest, Section 2.2.7) has several parameters for the construction algorithm and the data structure itself. The original paper does unfortunately not give values for all parameters. Some parameters are hard-coded in the implementation provided by the authors, while our implementation is very flexible and allows an easy tuning of the parameter values. We try to find good values for all parameters, so that the index works satisfactory in different settings (e. g., for different text lengths and alphabet sizes). We especially focus on longer texts here since the data structure is designed to be used in external memory.
The suffix array constructions take place in main memory so we use the algorithm of Larsson and Sadakane [LS07] (TAlgorithm = LarssonSadakane) by default because it is fast for various types of texts. The size of the partitions influences the performance in the sorting phase (2nd phase) but also in the merging phase (3rd phase), because more suffix arrays have to be merged when using shorter partitions. Our first guess was therefore that it is fastest to use preferably long partitions, but such that the resulting suffix arrays can be built in main memory. However, we found that rather short partitions give a faster overall index construction because the suffix array construction time grows slightly super-linearly with the partition size. For texts of different lengths n we tried several partition sizes and found out that a good rule of thumb is to use PARTITION_SIZE $= 1024 \cdot \sqrt{n}$ (e. g., 4 194 304 for $n = 2^{24}$ and 33 554 432 for $n = 2^{30}$ turned out to

be fast). However, the performance of the construction algorithm is quite robust regarding this parameter as long as the suffix array construction fits entirely into main memory.

The length of the tail appended to each partition does not influence the construction time much but is to ensure the correct sorting of the suffixes. Our implementation is designed so that wrongly sorted suffixes are detected and ignored, but reported to the user (the construction does not fail and all search queries that do not include the wrongly sorted suffix will still work correctly). We tested all test instances and found that `TAIL_LENGTH = 1000` works for all texts except for `text-english-28` (due to the long repetition mentioned in Section 6.3.3) and `fibonacci` (which contains or rather *consists of* very long repetitions and therefore cannot be used with the DiGeST construction algorithm).

The binary prefixes stored in the suffix arrays during the algorithm aim to reduce the costs of the comparison by increasing locality of reference. The longer the binary prefix, the faster the comparisons, because the underlying text does not need to be accessed often. The space consumption, however, also grows with the length of the prefixes. The prefixes are stored in a fixed number of machine words and the prefix length in bit can be controlled using the parameter `PREFIX_LENGTH`. In the original implementation of the data structure by Barsky et al. [Bar+08], the prefix length is fixed to 64 bit (two machine words). The 32 bit of one machine word correspond to x text characters depending on the underlying alphabet (e. g., $x = 4$ for `char`, 16 for `Dna`, 32 for `bool`, 5 for `AminoAcid`, and 1 for `wchar_t`)[8], and the text only has to be accessed in a comparison if all characters stored in the prefix were equal. In our implementation the binary prefix can span arbitrarily many machine words. For a uniformly distributed text it can be expected that *no comparison* needs to fall back on the text if $\sigma^x \geq n$, where x is the number of characters stored in the binary prefix. For a non uniformly distributed text this observation can be extended by using a measure for the actually used alphabet size, like the inverse probability of matching (Section 5.2.2), or by counting x-grams. However, the goal is only to reduce the number of comparisons and not to evade them entirely. In our experiments it turned out that 32 B are sufficient and result in a fast construction for most of the test instances. A prefix length of 64 should be chosen for texts using `char` with $n \geq 2^{30}$ and for all texts using `wchar_t` independently of the length.

The number of partitions is $\lceil n / \texttt{PARTITION\_SIZE} \rceil$ and the same number of suffix arrays is merged. For each suffix array an input buffer of `INBUF_SIZE` elements (consisting of the text position and a binary prefix of `PREFIX_LENGTH` bit) is reserved in main memory. In our experiments it turned out that `INBUF_SIZE` $= 65\,536 = 2^{16}$ works well in practice. The value should, however, be chosen so that the total input buffer size does not exceed the available main memory.

The size of the resulting partial suffix trees can be controlled using the parameter `OUTBUF_SIZE` (counted in terms of internal nodes). The original proposal by Barsky et al. [Bar+08] does not give a value for this parameter and for the performance of the search algorithm. They, however, state to use 6500 partial trees for the human genome of size ≈ 3 GB, which would correspond to partial trees with $\approx \frac{3 \cdot 10^9}{6500} \approx 460\,000$ leaves and therefore $\approx 920\,000$ internal nodes each. Based on that, we used the rounded value `OUTBUF_SIZE` $= 1\,048\,576 = 2^{20}$ as starting point for this parameter (resulting in trees of size 18 KiB). The value does not have a big influence on the construction time, but a smaller value generally leads to a slightly faster construction because the suffixes are inserted into smaller partial trees. It furthermore turned out later that the *search time* is dominated by the I/O costs to load the partial trees into main memory; the time to search in *dividers* and to descend in the partial tree is nearly negligible. An exact search can be carried out faster if the partial trees are smaller (there is no use in loading a bigger tree even if it is done with the same

[8]It is therefore crucial to select the correct alphabet type because when using, e. g., `char` instead of `Dna` for DNA sequences, the binary prefix only stores $\frac{1}{4}$ of the otherwise possible prefix length.

sequential disk access) and we therefore use a rather small value of `OUTBUF_SIZE = 1024` for the further experiments.

We built the DiGeST index for all test instances and observed that the performance degrades drastically when long exact repetitions are contained in the text, as for `text-english` and `fibonacci`. This is very similar compared to the WOTD and STTD64 suffix trees, because our construction algorithm of the DiGeST index needs to compute longest common prefixes of lexicographically neighboring suffixes, which is more expensive if there are long repetitions. For the other texts, the construction time is relatively similar (Figure 29).
The construction time for longer texts grows in practice linearly with the text length (as opposed to the theoretical time $\mathcal{O}(n \log n)$), see Figure 30. It can be built on our computer for texts of length up to 2^{30} by using rather big partitions and small input buffers to not exceed the little available main memory in the merging phase.
For longer texts, substantial engineering and performance profiling of the construction algorithm were necessary to identify bottlenecks and optimization possibilities (for example, we counted the number of *expensive* comparisons, i. e., comparisons that cannot be decided by using only the stored binary prefix etc.). In our experiments, the time for the partitioning phase (1st phase) was nearly negligible, about half the construction time is used for the sorting phase (2nd phase), and the other half is used for the merging phase (3rd phase).
It is interesting to note that our construction of a *suffix forest in external memory* can be even faster than the construction of a simple *suffix array* (e. g., for `dna-human-28`: 734 s vs. 795 s).

We compared our implementation to the original implementation provided by the authors [Bar+08]: in one hour we can build the index for `uniform-dna` of size 1.0 GiB and they report to build in about the same time the index for a similar text of size 1.8 GB = 1.68 GiB. Their computer has a faster CPU (2.66 GHz with two cores vs. our 2.15 GHz with one core) and more main memory (2.0 GiB vs. 1.4 GiB). This indicates that our implementation of the construction algorithm is competitive. The input in the original implementation comes in a compressed form (4 DNA bases stored in 1 B), so that the text only needs $\frac{1}{4}$ the size and more memory is available for the construction. We plan to use this simple input compression in our construction algorithm as well, the basic functionality is already offered by the underlying software library.

Recommendation for index parameters of `IndexDigest`:

1. `OUTBUF_SIZE`: A value of 1024 yields a good performance for both, the construction algorithm and the search algorithm.
2. `PREFIX_LENGTH`: For most texts 32 bit are sufficient, but 64 bit should be chosen for long `char` texts ($n \geq 2^{30}$) and for `wchar_t` texts.

Recommendation for construction parameters of `IndexDigest`:

1. `PARTITION_SIZE`: A good rule of thumb is to choose a value of $1024\sqrt{n}$ (e. g., for $n = 2^{24}$ use 4 194 304 = 2^{22} and for $n = 2^{30}$ use 33 554 432 = 2^{25}.
2. `TAIL_LENGTH`: For most real world texts a value of 1000 should be sufficient to guarantee a correct sorting.

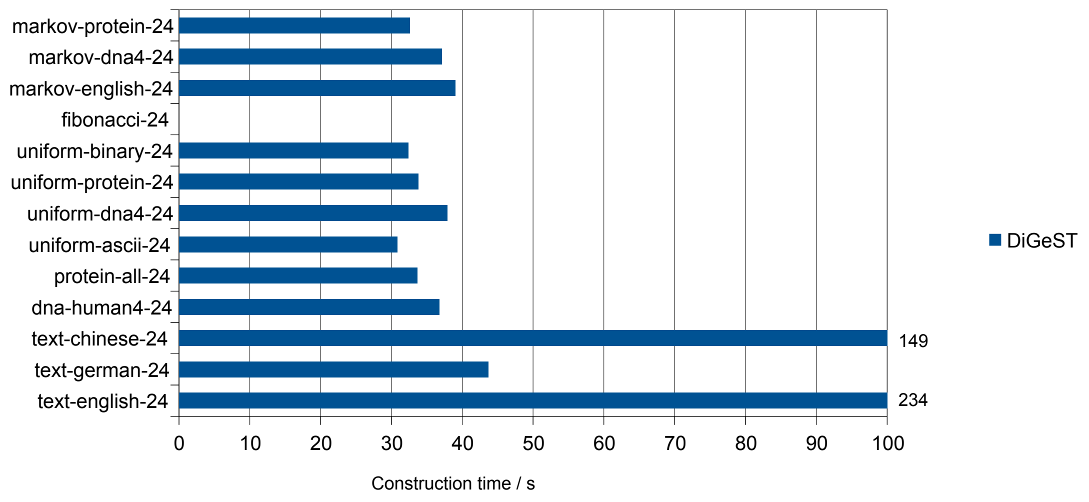


Figure 29: DiGeST index: construction time for different texts.

(Using $n = 2^{24}$ and the following parameters: `PARTITION_SIZE = 4 194 304`, `OUTBUF_SIZE = 1 048 576`, `INBUF_SIZE = 16384`, `TAIL_LENGTH = 1000`, `PREFIX_LENGTH = 32`)

`text-english-24`: The performance degrades if the text contains long repetitions, as discussed on page 136.
`text-chinese-24`: The algorithm works on the bit representation of the text and is therefore slower if wide characters (each 32 bit) are used. We furthermore use `PREFIX_LENGTH = 64` to achieve faster comparisons.
`fibonacci-24`: The DiGeST index does not work for strings that have exact repetitions of length > `TAIL_LENGTH` spanning two or more partitions, and the Fibonacci string contains or rather *consists of* very long repetitions.

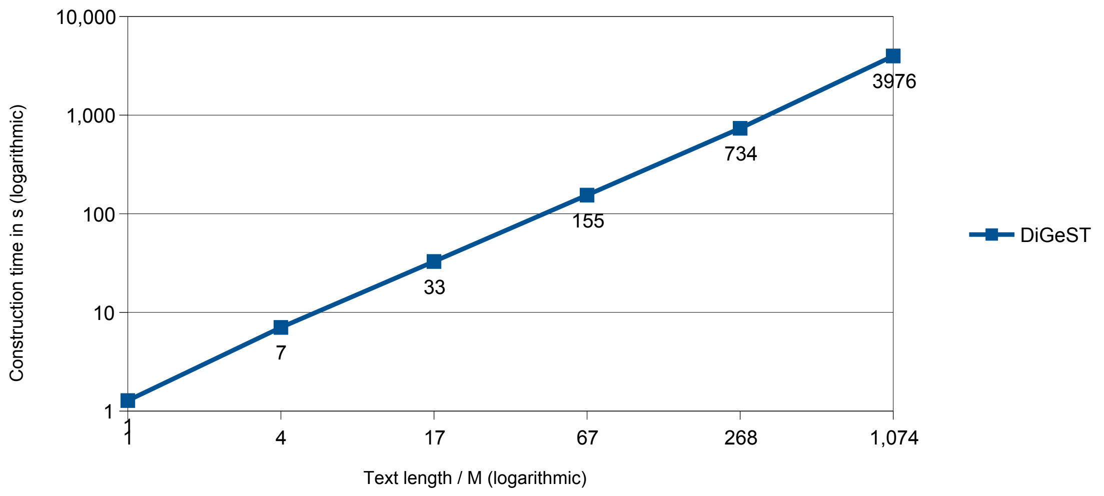


Figure 30: DiGeST index: construction time depending on the text length.

(Using t = `uniform-dna4`, $n \leq 2^{30}$ and the following parameters: `PARTITION_SIZE = 8 388 608`, `OUTBUF_SIZE = 1 048 576`, `INBUF_SIZE = 16 384`, `TAIL_LENGTH = 1000`, `PREFIX_LENGTH = 32`, and deviating for $n = 2^{30}$: `PARTITION_SIZE = 33 554 432`, `INBUF_SIZE = 65 536`)

The general picture and also the absolute values are very similar also for, e. g., `dna-human4` and `uniform-ascii`.

3. `INBUF_SIZE`: $65536 = 2^{16}$ works well in many scenarios, but a lower value should be used if the number of partitions is high.

6.3.7 FM index

The performance of the FM index (`FMIndex`, Section 2.3.1) is mainly dependent on the compression rate which offers a trade-off between index space and search time. We tried different values between 1 and 256 and recorded the construction time and space usage. The construction time is basically independent of the compression factor, but the index space decreases quickly with higher values, see Figure 31. For a compression factor of 16, the space of the resulting index is already very low, between $0.6\,n$B (for the binary sequences `uniform-binary` and `uniform-binary`) and $2.0\,n$B (for `uniform-ascii`). We therefore use `compressionFactor=16` as default value for the further experiments.
We constructed the FM index for all test sets of length 2^{24} using both implementation options for the occurrences table (wavelet tree `WT` and sequence bit masks `SBM`), see Figure 35. It turns out that the construction time of both options are very similar because the construction is dominated by the time to build the suffix array (`WT` takes in total about 10 % longer on average). The construction space is also dominated by the space to build the suffix array.
The space consumption of the resulting data structures depends heavily on the type of alphabet of the underlying text, see Figure 36. For larger alphabets ($\sigma > 5$), the wavelet tree variant needs less space, for smaller alphabets, the sequence bit masks are favorable. Texts that use a big alphabet in general result in a bigger index, which is also explained by the theoretically linear dependence of the space on the entropy of the text (visualized in practice in Figure 32). We can also observe a correlation of the practical compressibility of a text and the size of the resulting FM index (Figure 33).

The construction algorithm of the FM index first builds the uncompressed suffix array and so the maximal text length is limited by this algorithm and its space usage (which is several times the size of the resulting index). We were able to build the FM index for texts up to length 2^{28} and the construction algorithms shows a slightly super-linear dependence on the text length (Figure 34).
A detailed experimental analysis of the implementation of the FM index is given by Singer [Sin12]. The results show in particular that the given implementation of the software library outperforms the original implementation of the authors of the index.
Our results for longer texts indicate that the implementation of the software library is faster than the version used, e. g., by Hon et al. [Hon+04]: For a DNA sequence of size 119 MiB they report to need 36 min and the library's implementation needs only 14 min for a DNA sequence of size 256 MiB. Their computer has a slower CPU (1.7 GHz vs. our 2.15 GHz) but more main memory (4.0 GiB vs. 1.4 GiB).

Recommendation for index parameters of `FMIndex`:

1. `TOccSpec`: Should be chosen depending on the alphabet size σ: `SBM` for $\sigma \leq 5$ and `WT` for $\sigma > 5$.
2. `compressionFactor`: Allows a trade-off between index size and search time (see also Figure 31). A value of 16 yields an index size in the order of the text length (higher for large alphabets). Does not significantly influence the construction time, but the search time increases approximately linearly.

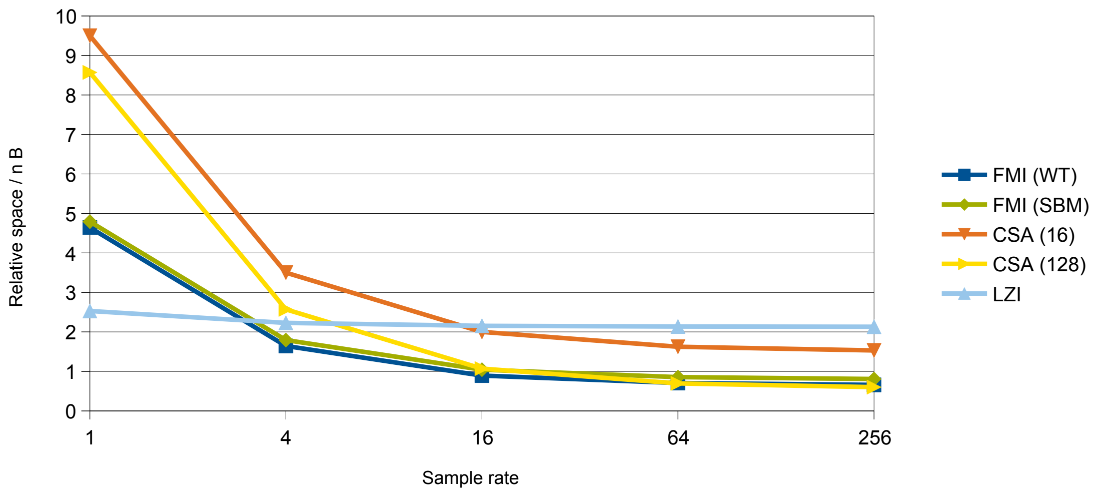


Figure 31: Compressed indexes: relative space for different sample rates.

(Using t = `uniform-dna-24`.)

The *sample rate* denotes for the FM index the compression factor, for the CSA the sample rate of the suffix array and inverse suffix array (both set to be equal), and for the LZ index the text position sample rate.
(For the CSA we use two variants: `PSI_SAMPLE_RATE=16` and `128`, indicated in parentheses.)
The picture is very similar for other kinds and sizes of text (regarding the relative behavior; the absolute values are different, see Figure 36).

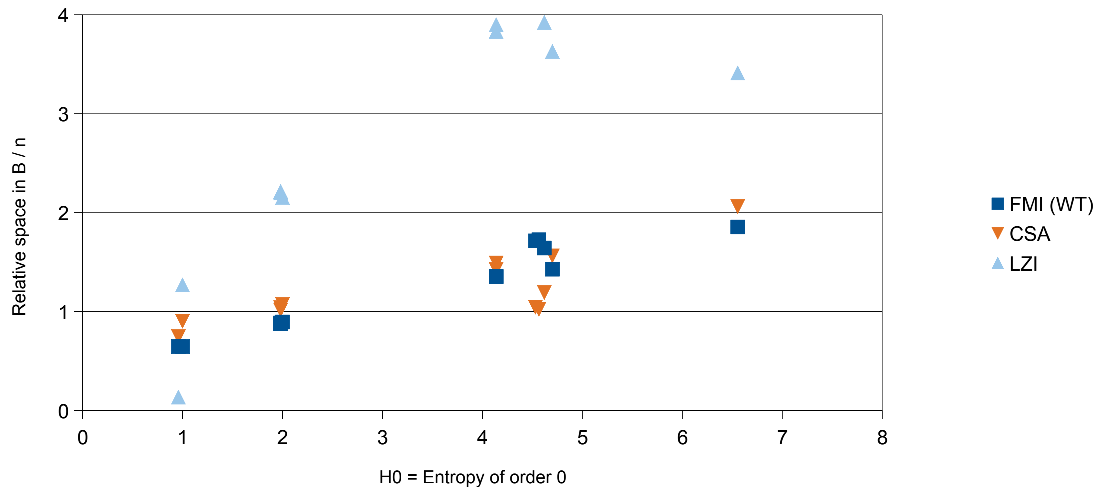


Figure 32: Compressed indexes: relative space depending on the text entropy.

(Using $n = 2^{24}$ and the following parameters for the FM index: `compressionFactor=16`, for the CSA: `PSI_SAMPLE_RATE=128, SA_SAMPLE_RATE=16, ISA_SAMPLE_RATE=16`, and for LZ index: `LZTRIE_NODE_END_TEXT_POSTION_SAMPLE_RATE=16`.)

Each dot represents a test instance and we can observe also in practice a roughly linear dependency of the index space on the empirical entropy of the text for FMI and CSA.

6.3.8 Compressed suffix array

The experiments regarding the compressed suffix array (IndexSadakane, Section 2.3.2) were performed analogously to the FM index. For the table Ψ we use a sample rate of PSI_SAMPLE_RATE = 128 which gives a good trade-off between search time and index space [Fer+09b; Aum11]. For the suffix array and the inverse suffix array we use the same sample rate, which helps us to reduce the size of the parameters space; we tried different values for SA_SAMPLE_RATE and ISA_SAMPLE_RATE between 1 and 256 (Figure 31). We observed that for a value of 16, the size of the resulting index is in the order of the length of the text (between 0.7 n B for fibonacci and 2.1 n B for uniform-ascii). In other experiments we measured the search time of the compressed suffix array using different sample rates and it turned out that a value of 4 significantly increases the search performance while the size of the index is more or less doubled compared to a sample rate of 16. We therefore use PSI_SAMPLE_RATE = 128, SA_SAMPLE_RATE = 4, ISA_SAMPLE_RATE = 4 as the default configuration for further experiments. The construction time is nearly independent of the sample rates.

We built the compressed suffix array for all texts of size 2^{24} and the construction times vary between the instances with a factor of up to 2 (Figure 35). The space usage for the test instances is shown in Figure 36. It is lower for small alphabets and higher for large alphabets, very similar to the FM index. The CSA is slightly bigger than the FM index, except for the natural language texts text-english and text-german and their synthetic counterpart markov-english, in which cases it is smaller. The necessary construction memory is between 8.7 and 10.1 n B.
We can also see a nearly linear dependency of the relative index size (measured in B per text character) on the entropy of the text (Figure 32). We can observe a linear dependency of the relative index size and the relative compressed size of the text (Figure 33).

We were able to build the CSA with the direct construction algorithm (without incremental creation) for texts up to length 2^{26} in main memory. For longer texts we have to switch to the incremental construction. We tried several values for the block length and it turned out that the highest possible value where all data structures still fit into main memory is fastest. Therefore, the block length should be smaller for longer texts, since less memory remains available for the construction. For texts of length 2^{28} we therefore use a block length of 32 MiB (which results in a peak virtual memory usage of 1466 MiB which just fits into the main memory of our computer). The incremental construction comes at the cost of a significantly slower construction, see Figure 34: a linear extrapolation from smaller texts would give for $n = 2^{28}$ a construction time of about 1000 s, but due to the incremental construction it takes 2888 s.

The obtained results for longer texts indicate that our implementation of the construction algorithm mainly done by Stadler [Sta11] is competitive to the original implementation by Hon et al. [Hon+04]: For a DNA sequence of size 119 MiB they report to need 30 min and our implementation needs 4 min for a DNA sequence of size 64 MiB and 48 min for 268 MiB. Their computer has a slower CPU (1.7 GHz vs. our 2.15 GHz) but more main memory (4.0 GiB vs. 1.4 GiB).

Recommendation for index parameters of IndexSadakane:

The parameters allow the choice between a small index and a fast search time. The following values provide a good trade-off (see also Figure 31):

1. PSI_SAMPLE_RATE: 128
2. SA_SAMPLE_RATE: 4

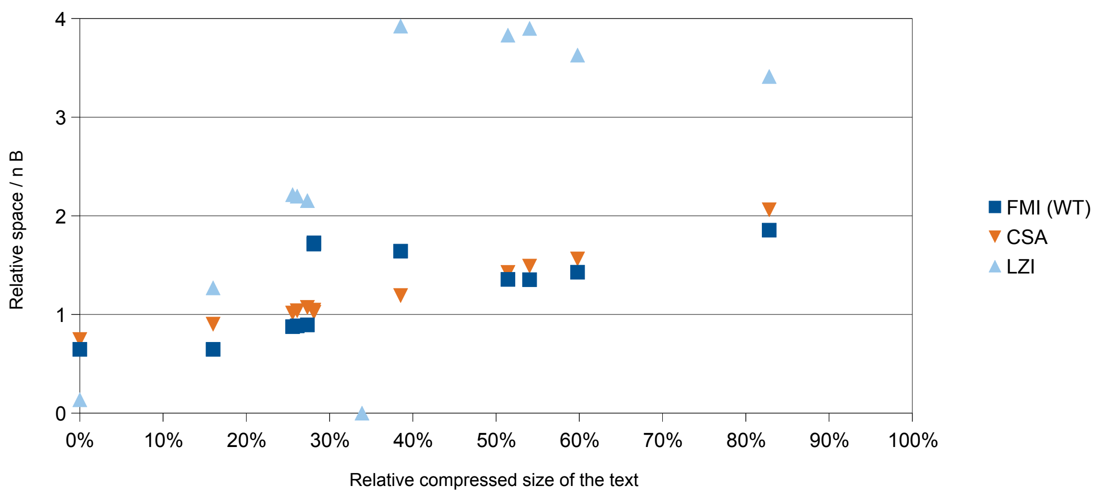


Figure 33: Compressed indexes: relative space depending on the text compressibility.

(Using $n = 2^{24}$ and the same index parameters as in Figure 32.)

Each dot represents a test instance and we can observe a roughly linear dependency of the index space on the relative compressed size of the text for FMI and CSA.
As compressor we use `bzip2` here. The picture is similar for `gzip` and `xz`, but the linear correlation is less marked in these cases.

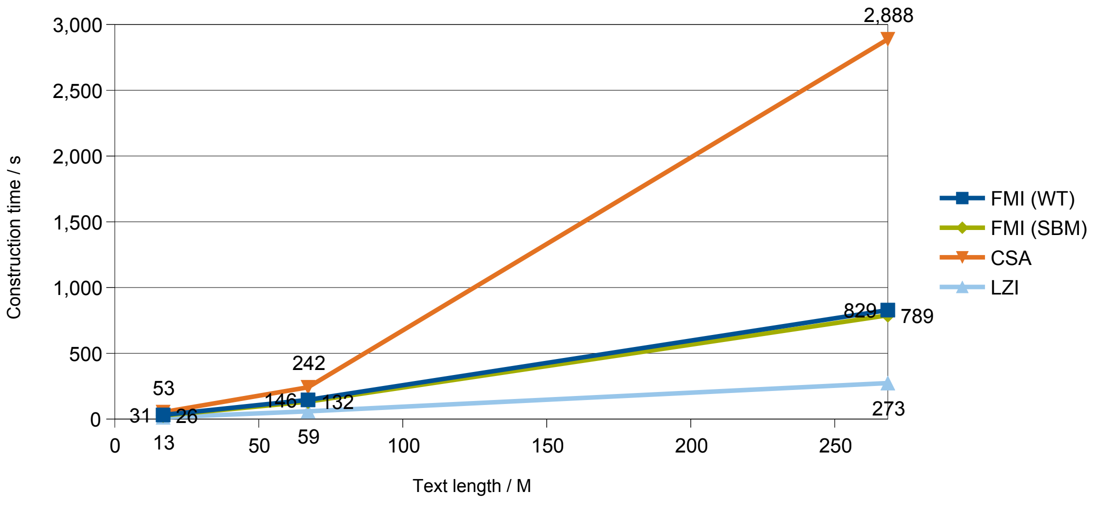


Figure 34: Compressed indexes: construction time depending on the text length.

(Using t = `uniform-dna4`, $n \leq 2^{28}$ and the same index parameters as in Figure 32.)

The CSA construction fits into main memory for texts up to $n = 2^{26}$. For longer texts we have to switch to the incremental construction, here using a block length of 32 MiB.

3. ISA_SAMPLE_RATE: 4

Recommendation for construction parameters of IndexSadakane:

1. blockLength: The block length should be as long as possible, so that all data structures of the construction still fit into main memory.

6.3.9 LZ index

The third compressed index of the implementation is the LZ index (IndexLZ, Section 2.3.3). The parameter for the sample rate turned out to have only little influence on the overall size of the resulting index, in our experiments it made a difference of at most 0.5 n B between LZTRIE_NODE_END_TEXT_POSTION_SAMPLE_RATE = 1 and = 256 (Figure 31). The search time, however, grows approximately linearly with the sample rate and we therefore use a rather small value of 4 in the further experiments.
The construction time varies significantly between the different texts: for the very repetitive text `fibonacci-24` it takes only 2.3 s and for `uniform-ascii` with hardly any long repetitions it takes 67.4 s. Similar observations hold for the index space usage which is smallest for the binary sequences (0.14 n B for `fibonacci-24`) and biggest for the natural language texts (4.2 n B for `text-german-24`), see Figure 36. However, we could not observe a strong linear correlation of the index size on the entropy or the relative compressed size of the text, using different compressors (Figure 32 and Figure 33). The construction space lies between 1.3 n B and 13.7 n B (but is for all *real world* instances below 9 n B).

We were able to construct the LZ index for text up to 2^{28} and the construction algorithm turns out to be very fast compared to the algorithms of the other compressed indexes: 3 times faster than for the FM index and 10 times faster than for the CSA (Figure 34). The theoretical construction time of $\mathcal{O}(n \log \sigma)$ matches the results of our experiments (Figure 35 together with Figure 34).

The obtained results for longer texts indicate that our implementation of the construction algorithm mainly done by Stadler [Sta11] is slightly slower than the implementation used by Ferragina et al. [Fer+09a]: For an English text of size 191 MiB they report to need 3.3 min and our implementation needs 8.3 min for an English text of size 256 MiB. Their computer has a slightly faster CPU (2.6 GHz vs. our 2.15 GHz) and comparable main memory size (1.5 GiB vs. 1.4 GiB).

Recommendation for index parameters of IndexLZ:

1. LZTRIE_NODE_END_TEXT_POSTION_SAMPLE_RATE: The sample rate does not significantly influence the space (see also Figure 31), so a small value can be chosen to provide a faster search (default: 4).

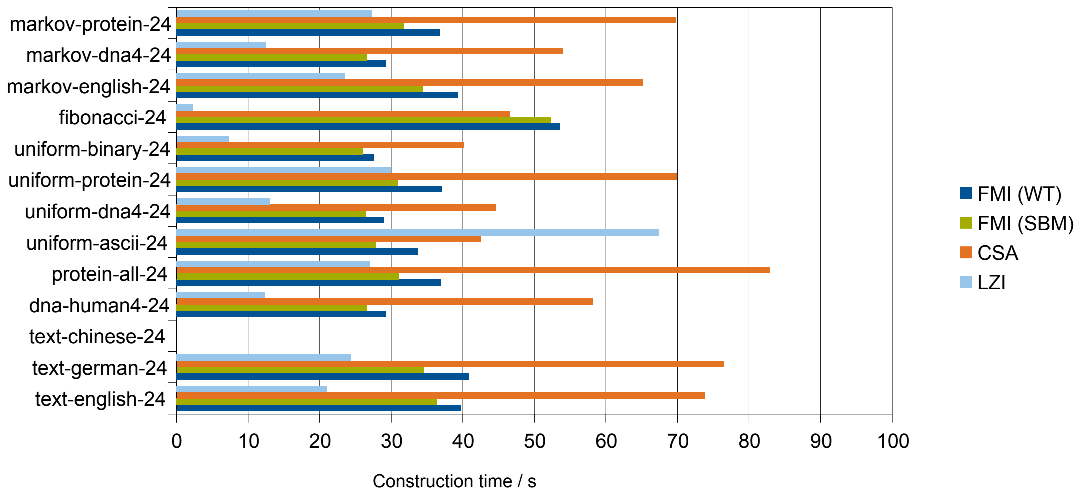


Figure 35: Compressed indexes: construction time for different texts.

(Using $n = 2^{24}$ and the same index parameters as in Figure 32.)

`text-chinese-24`: The implementations do not work for wide characters.

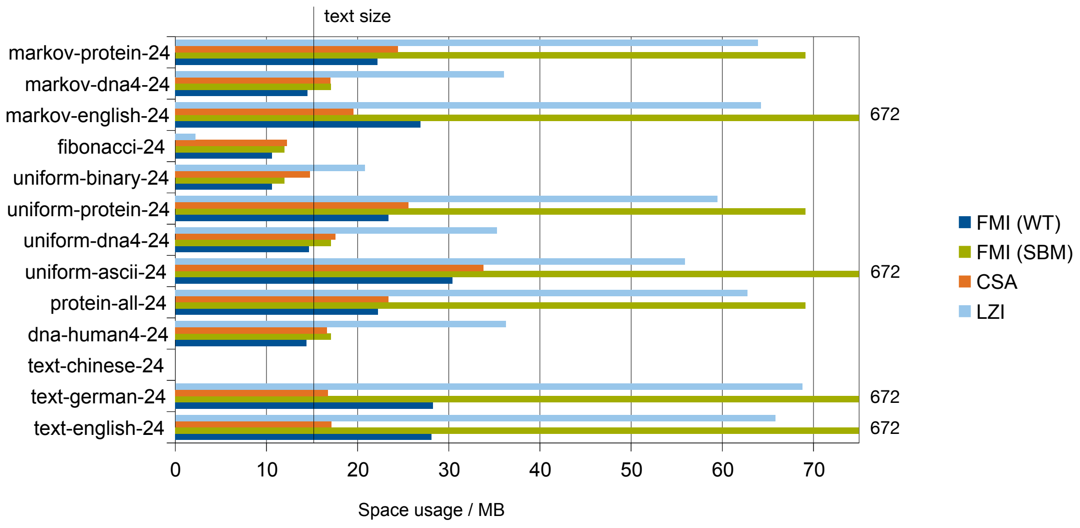


Figure 36: Compressed indexes: space usage for different texts.

(Using $n = 2^{24}$ and the same index parameters as in Figure 32.)

6.3.10 q-gram index

The most crucial parameter of the q-gram index (`IndexQGram`, Section 2.4.1) is certainly q, the length of the indexed substrings. We measured the impact on the construction time and space usage for values of q between 2 and 12 for all test instances. For low values of q, the construction is fastest and for growing q it eventually gets significantly slower; for which value of q this happens depends on the size of the alphabet (or rather the size of the data type used to store the characters): with `Dna` sequences this is for $q \geq 12$, with protein sequences (`AminoAcid`) for $q \geq 5$ and for `char` based sequences already for $q \geq 3$ (Figure 38). This behavior can best be explained when analyzing the space consumption of the index (Figure 39): For greater values of q, the size σ^q of the *directory* table eventually "explodes" compared to the other data structures and so the construction takes much more time. Below that point, the q-gram index takes essentially only $4\,n\,\mathrm{B}$ because the space is dominated by the *positions* table.

The q-gram can use a direct addressing and open addressing. We observed that the construction is fastest if we stick to the direct addressing scheme as long as the resulting *directory* table (σ^q entries) fits into main memory. (Direct addressing is also advantageous because it permits to search for patterns shorter than q, see Section 2.4.1; otherwise, the search performance is nearly independent of the addressing scheme.) For greater values of σ and q we should (or rather have to) use open addressing. The size of the open addressing table then depends on the length of the text (by default $1.6n$ entries) and is independent of the alphabet size σ and of q; this is illustrated in Figure 39.

We tried to find a relation between properties of the underlying text and the construction time since we noted that the times vary with a factor of 20 (for $q = 3$ and texts of length 2^{24}). We found that the alphabet size can only give an indication, but the actual number of different q-grams can better be used for an approximation of the necessary construction time of a q-gram index (using a roughly logarithmic dependency as can be observed in Figure 37).

The *directory* table should always be available in main memory since it is accessed randomly. The *positions* table can be stored in external memory and we found a page size of 1 MiB works well. However, we observed that the performance of the construction algorithm degrades as soon as there are less than $\sigma^q + 1$ frames reserved for the *positions* table (since it is accessed at $\sigma^q + 1$ different positions).
The construction time increases linearly with the text length and we were able to build the q-gram index also for texts of length 2^{30} (Figure 40) using external storage as described above. This, however, only works for limited values of q (depending on σ) since $\sigma^q + 1$ frames should be available in main memory for the *positions* table. Using the rather small page size of 16 KiB we were able to build the q-gram index for $q = 6$ and `uniform-dna4` of length 2^{30} on our computer in 890 s $\approx$ 15 min.

(The value of q is limited by $\sigma^q < 2^{64}$ due to the limited size of the hash values (64 bit). If the q-gram index is used beyond this point, we risk integer overflows and as a consequence collisions, i. e., different q-grams that have the same hash value. We noticed that in practice the index still works correctly for higher q if a subsequent verification step is implemented. We think this could be a practically feasible solution to permit greater values of q but we do not use this trick here.)

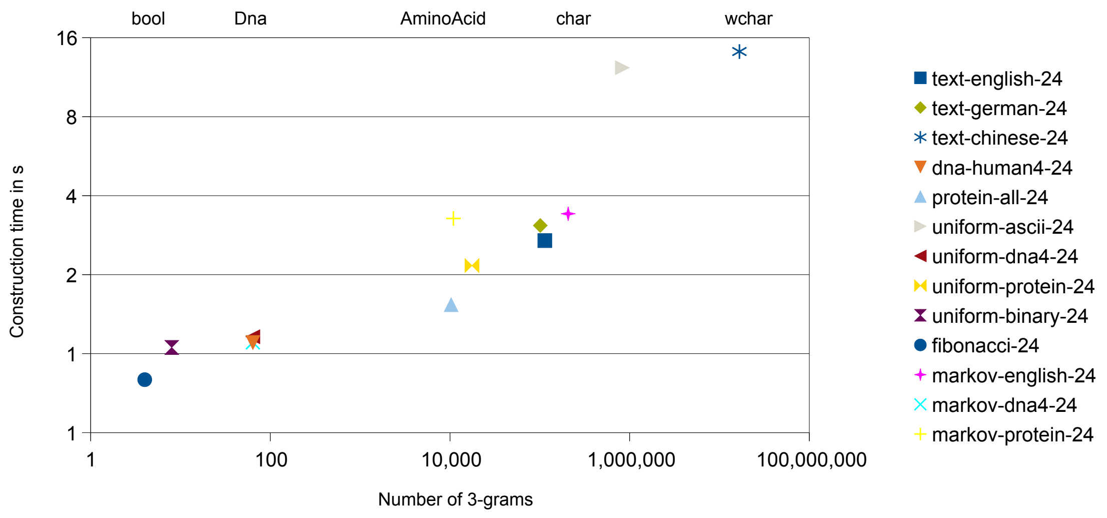


Figure 37: *q*-gram index: construction time depending on the number of different *q*-grams.

(Using $n = 2^{24}$, $q = 3$, in-memory storage `Alloc` and when necessary open addressing.)

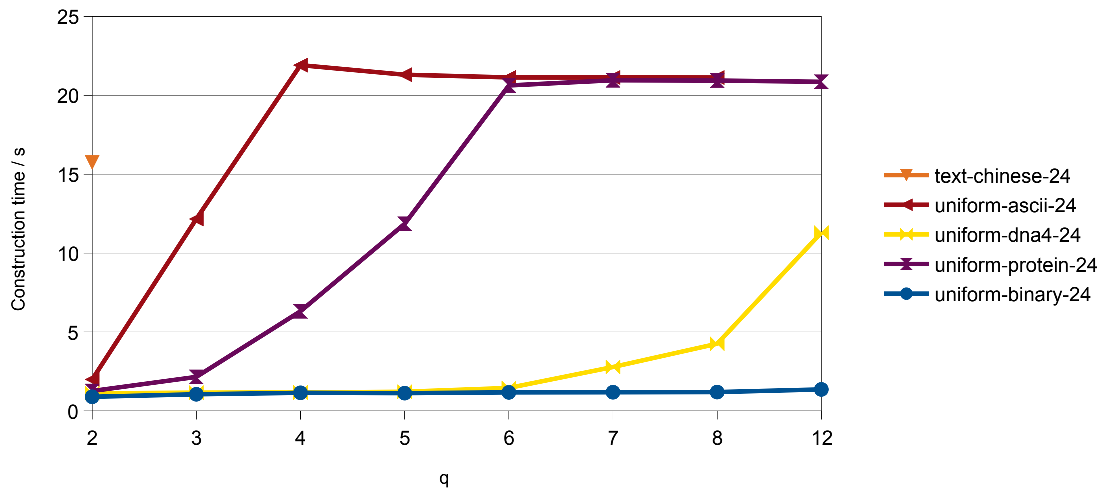


Figure 38: *q*-gram index: construction time depending on *q* for different texts.

(Using $n = 2^{24}$, in-memory storage `Alloc` and when necessary open addressing.)

We give the results only for one representative of each alphabet type since the other texts give very similar results.
`text-chinese-24`: Texts using `wchar_t` characters are limited to $q \leq 2$ (see Section 2.4.1).
`uniform-ascii-24`: Texts using `char` characters are limited to $q \leq 8$ (see Section 2.4.1).

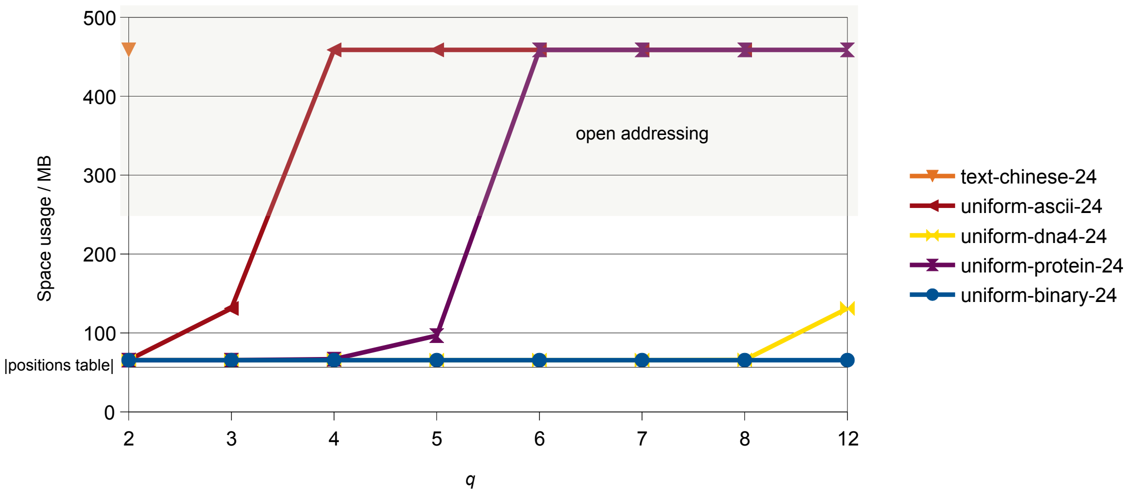


Figure 39: *q*-gram index: space usage depending on *q* for different texts.

(Using $n = 2^{24}$, in-memory storage `Alloc` and when necessary open addressing.)

We give the results only for one representative of each alphabet type since the other texts give very similar results.
`text-chinese-24`: Texts using `wchar_t` characters are limited to $q \leq 2$ (see Section 2.4.1).
`uniform-ascii-24`: Texts using `char` characters are limited to $q \leq 8$ (see Section 2.4.1).

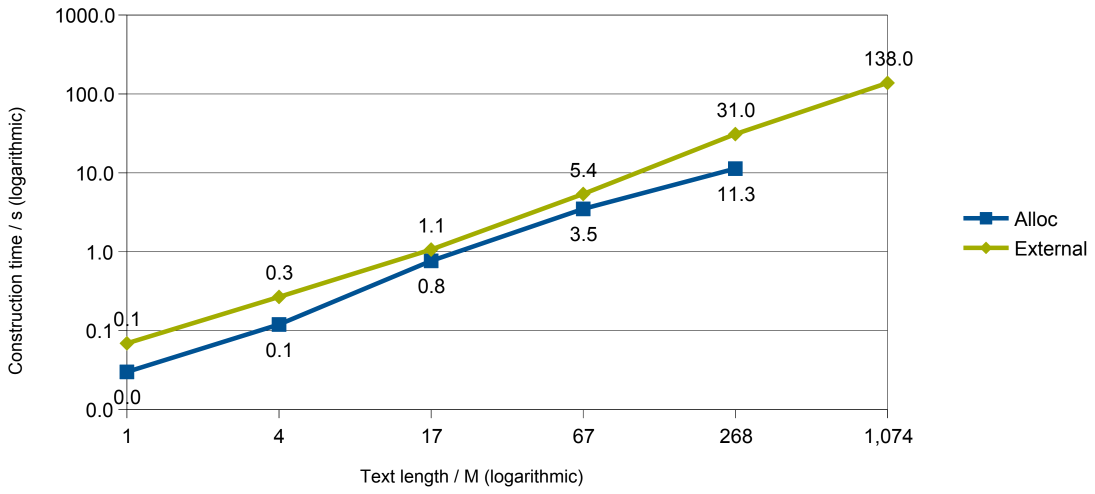


Figure 40: *q*-gram index: construction time depending on the text length.

(Using t = `uniform-dna4`, $n \leq 2^{30}$, and $q = 3$ with direct addressing.)

The construction time grows linearly with the text length and the main memory variant is generally fast. For indexes exceeding the available memory (here for $n = 2^{30}$), the external memory variant has to be used.

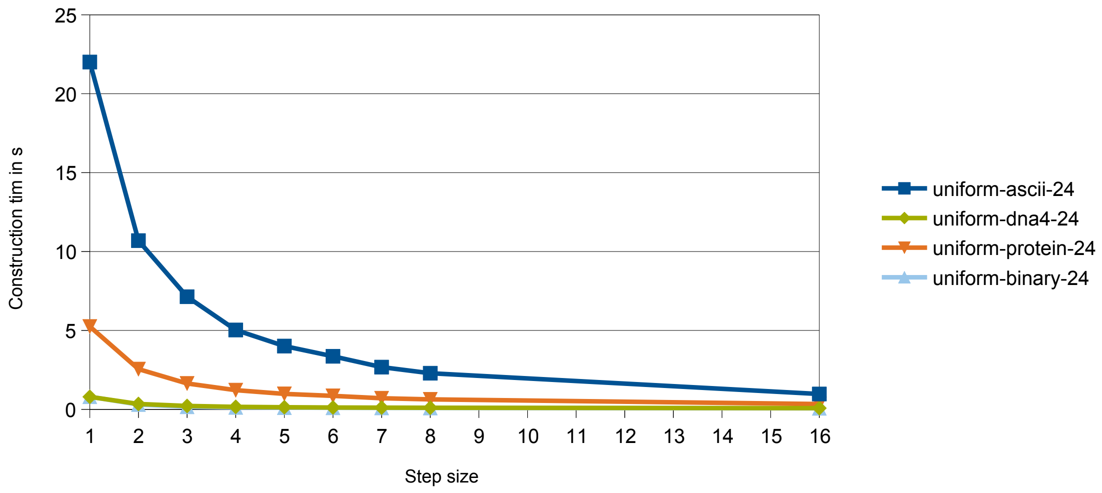


Figure 41: *q*-sample index: construction time depending on the step size.

(Using $n = 2^{24}$, $q = 4$, and storage in main memory with `Alloc`. We use direct addressing for all texts, except for `uniform-ascii-24` where we have to use open addressing.)

The line for `uniform-binary-24` is overlapped by `uniform-dna-24`.

The picture is very similar for the space usage.

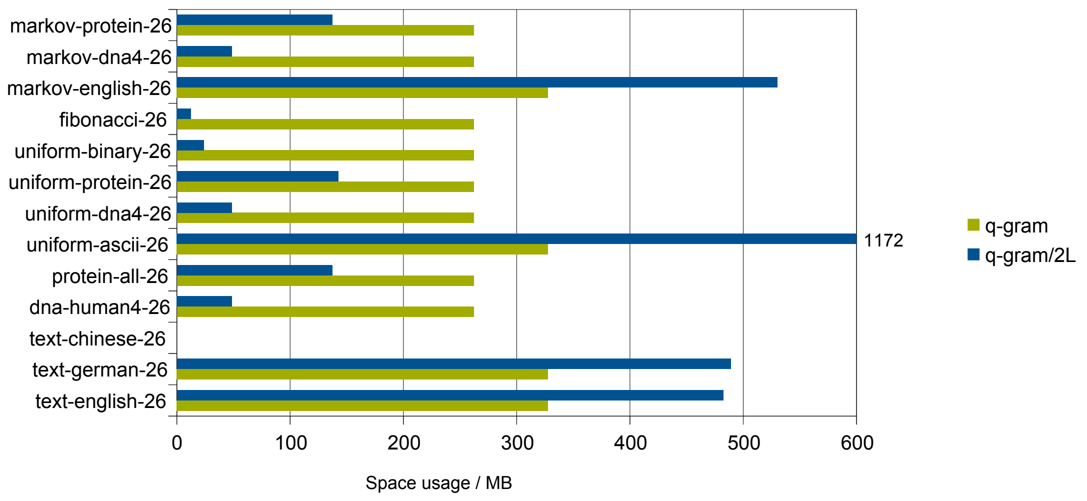


Figure 42: *q*-gram and *q*-gram/2L index: space usage for different texts.

(Using $n = 2^{26}$, $q = 3$ and the respective best subsequence length *h*, see Figure 43.)

The space reduction of *q*-gram/2L is most remarkable for small alphabet sizes and, as a consequence thereof, for texts with a small number of different *q*-grams (and *h*-grams respectively).

The test instance `text-chinese` is shorter than 2^{26} and therefore not tested here.

Recommendation for index parameters of `IndexQGram` (classic):

1. `Q`: Should be chosen as big as possible, so that the resulting index fits into main memory.
2. `TSpec`: Use direct addressing (`void`) if the *directory* table of size σ^q fits into main memory and `OpenAddressing` otherwise.
3. `TIndexStringSpec`: The *directory* table should always be kept in main memory. If the *positions* table does not also fit into main memory (`Alloc`) external storage (`External`) should be used and at least $\sigma^q + 1$ frames should be reserved. A page size of 1 MiB is a good choice, but use a lower value if otherwise the frames do not fit into main memory; it should hold: $\mathit{pageSize} \cdot (\sigma^q + 1) <$ available memory.

6.3.11 *q*-sample index

The *q*-sample index (implemented in `IndexQGram`, Section 2.4.2) is a variant of the *q*-gram index and has an additional parameter for the step size. The space of the *positions* table is reduced by a factor *stepSize* compared to the classical *q*-gram index, while the size of the *directory* table stays the same. We noticed, however, that in practice the size of the *complete index* is reduced by this factor: Either the *directory* table is negligibly small compared to the *positions* table or is sufficiently big so that open addressing is used instead of direct addressing; the size of the open addressing hash table depends on the number of indexed *q*-grams and is therefore also reduced by a factor of *stepSize*.

The construction time is also reduced by this factor as can be seen in Figure 41. The overall behavior of the construction of the index structure is otherwise very similar to the classical, not sampled *q*-gram index.
The search time for an exact pattern matching query grows roughly linearly with increasing *stepSize* which, however, can be tolerated in many scenarios in exchange for a smaller index structure.

Recommendation for index parameters of `IndexQGram` (sampled):

4. `setStepSize()`: The step size allows a trade-off between a fast search (low step size) and a smaller index structure (higher step size). In any case, the restriction regarding the pattern length has to hold (Section 2.4.2).

6.3.12 *q*-gram index with two levels

The *q*-gram/2L index with two levels (`IndexQGram2L`, Section 2.4.3) offers the same interface as the classical *q*-gram index, but attempts to reduce the space usage. The value *h* defines the length of the indexed subsequences and it is crucial to choose an adequate value. The choice has to be made depending on *q* and on the characteristics of the underlying text (especially on the alphabet size). Choosing *h* too high results in a big back-end index and choosing *h* too low results in a big front-end index. We tested for all texts and $q \in [2, 8]$ different ranges of values for *h* and measured the space of the resulting index (here we optimize the parameter regarding the *space* because the search and construction times do not differ much depending on *h* and this index is

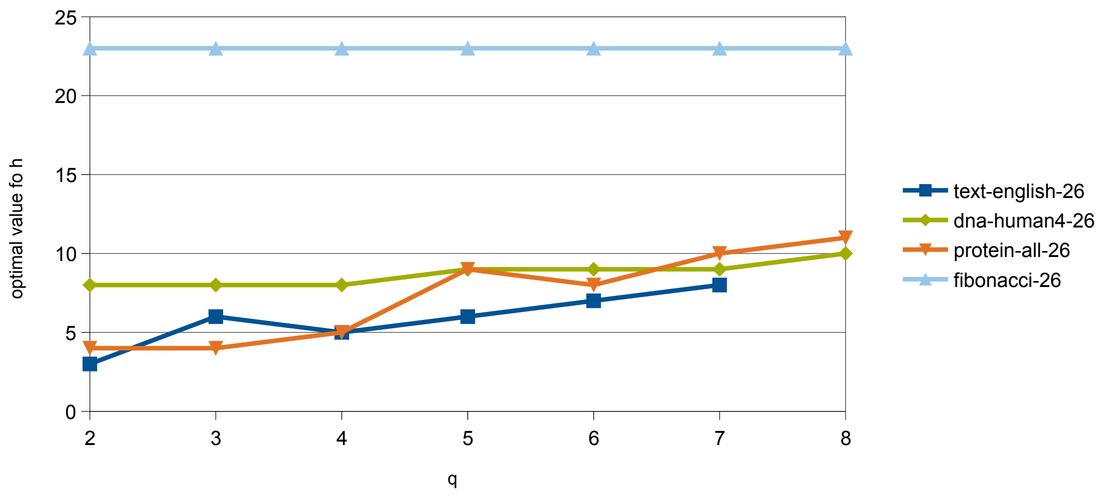


Figure 43: *q*-gram/2L index: optimal parameter value *h* depending on the gram size *q*.

(Using $n = 2^{26}$.)

For the data type `char` and $q = 8$, the *q*-gram/2L index is not applicable because the hash values of the subsequences (of length $h \geq 9$) do not fit into a 64 bit integer.

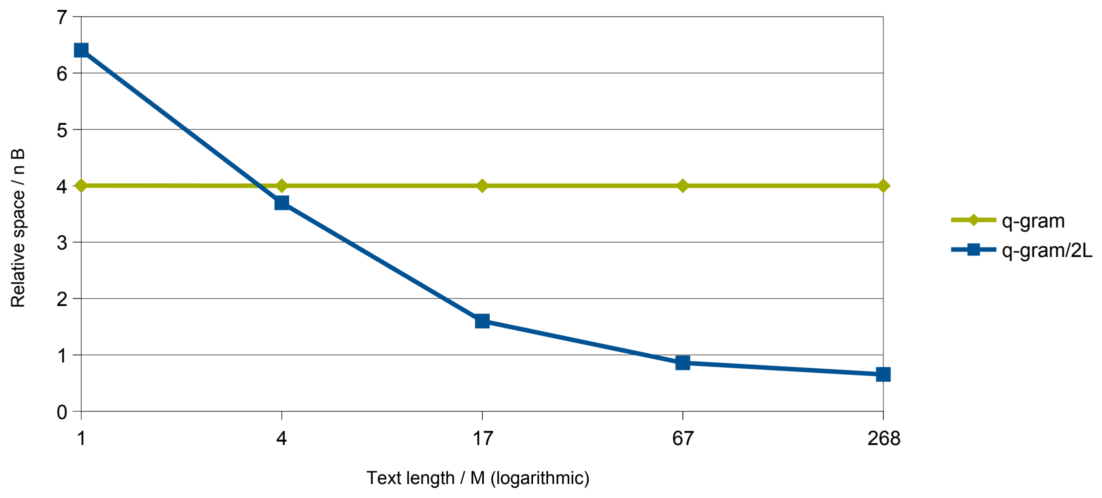


Figure 44: *q*-gram/2L index: relative space usage depending on the text length.

(Using t = `dna-human4`, $q = 3$, $h = 9$, and $n \leq 2^{28}$.)

The *q*-gram/2L index saves more space for longer texts (shown here for DNA sequences in comparison with the *q*-gram index).

targeted at reducing the size of the data structure). The results for $q = 3$ and all test instances are given in Figure 45; the optimal value for the subsequence length h varies between 4 and 23. The dependence of the optimal value for h on the q-gram size q is shown in Figure 43. We use these optimal values in further experiments if not indicated otherwise.

The addressing scheme (direct addressing or open addressing) should be chosen based on the required space for the *directory* table just as for the classical q-gram index.

The space of the q-gram/2L index in comparison with the classical q-gram index for all test instances is shown in Figure 42. The two-level structure can significantly save space compared to the classical q-gram index if the alphabet is small. The space is reduced, e. g., for DNA sequences and $q = 3$ from $4\,n$ B to $0.7\,n$ B. For larger alphabets (in our tests for `char` instances) and especially if the number of different q-grams occurring in the text is high, the q-gram/2L index is bigger than the classical q-gram index (e. g., for `uniform-ascii`).

The construction times are consistently similar to the classical q-gram index also for different q and text lengths (varying with a factor between 0.4 and 2.4). The construction space amounts to between 1.0 and 3.0 the size of the index and is therefore in only a few cases higher than the space needed for the classical q-gram index.
The authors of the original proposal by Kim et al. [Kim+05] do not give indications of their construction times. The space savings of our implementation are, however, very similar to the original implementation (they count the number of disk pages and we measure the space in byte).
A nice property of the q-gram/2L index is that the relative space usage (space per text character) decreases significantly with increasing text length, visible in Figure 44 in comparison with the space usage of the classical q-gram index. This is because the redundancy can be exploited even better if the subsequences occur more often.

Recommendation for index parameters of `IndexQGram2L`:

1. `Q`: Should be chosen depending on the application.
2. `TSpec`: `OpenAddressing`
3. `subsequenceLength`: The length h of the extracted subsequences should be chosen so that the space consumption is minimized. This depends on the actual properties of the underlying text. For DNA sequences $h = 8$ or $h = 9$, for natural language texts $h = q + 1$ (and $h = 6$ for $q = 3$), and for protein sequences $h = q + 1$ works well in most cases (see Figure 43).

6.3.13 Comparison

To compare all index structures regarding the space usage, construction space, and construction time, we present them exemplarily in two diagrams (Figure 46): one for natural language (German) texts and one for DNA sequences. This makes it easy to select an index structure based on the available memory and construction time. Note, however, that some index structures can be configured with parameters changing their performance. Furthermore, the diagram does not show all aspects since some index structures can, e. g., use an incremental construction or can efficiently be built in external memory as described above.

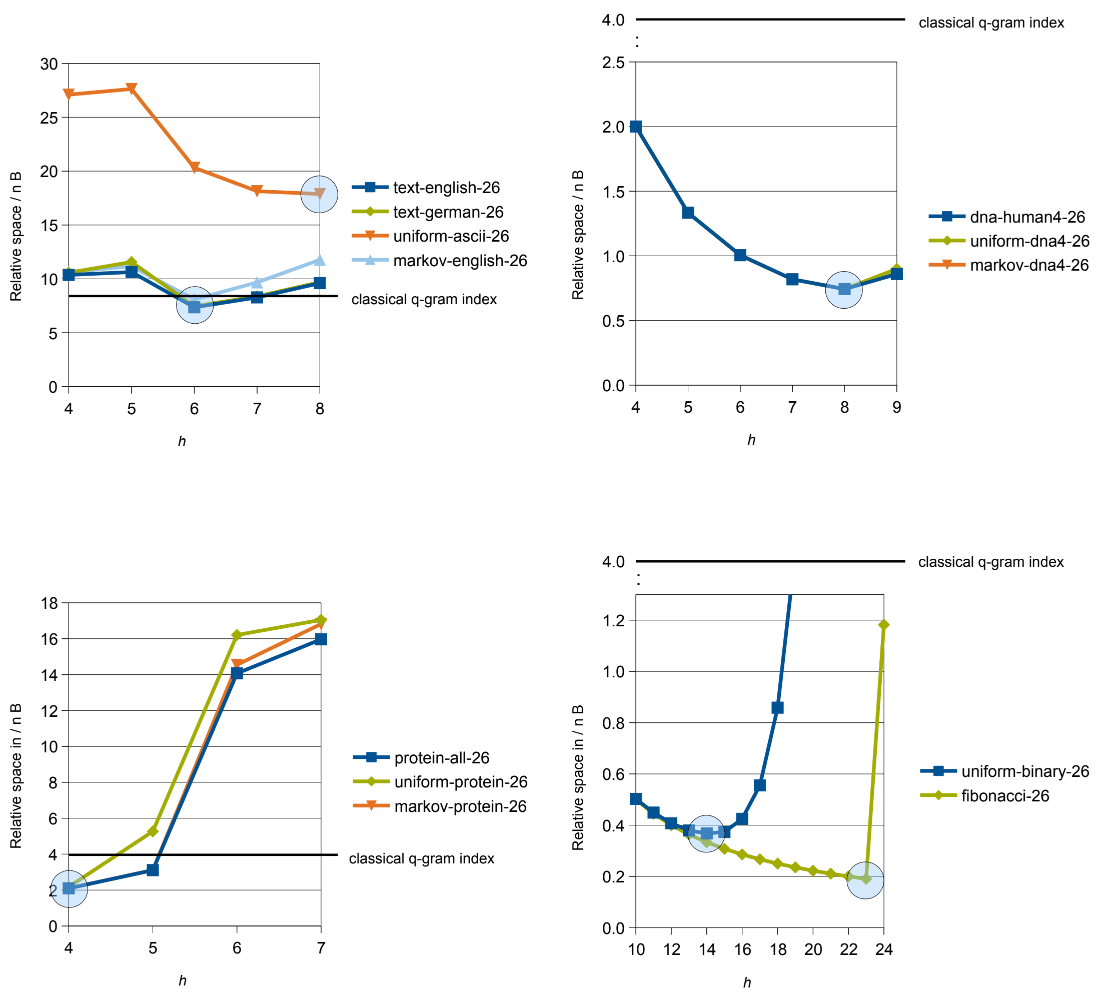


Figure 45: *q*-gram/2L index: finding the optimal value for the subsequence length *h*.

(Using $n = 2^{26}$, $q = 3$ and `TSpec=OpenAddressing`)

The respective best choice for the length *h* of the subsequences is marked with a circle (like in [Kim+05]). The behavior for texts using the same alphabet is rather similar.
The size of the classical *q*-gram index is indicated by the respective lines as comparison.

One interesting case is the Fibonacci string where the optimal subsequence length $h = 23$ turned out to much higher than for the other texts. This is because the subsequences are sampled from the text with a step size of $h - q + 1$, here $23 - 3 + 1 = 21$. This is a Fibonacci number and denotes the length of f_7, the 7th Fibonacci word (Section 5.3.2) which is contained repeatedly in the text. The order of 3-grams within this string therefore only has to be stored once, resulting in a very small index of size 0.2 *n* B only (compared to 4 *n* B of the classical *q*-gram index). Choosing higher Fibonacci numbers, e. g., $h = f_8 + 3 - 1 = 36$ or $h = f_9 + 3 - 1 = 57$ does not reduce the index size because of a growing back-end index.

(Some lines are not visible because they are overlapped by other lines.)

6.4 Algorithms for approximate search

We performed a systematic experimental analysis of algorithms for approximate pattern matching. At first, we summarize the results for online algorithms, which are, however, not the main focus of this thesis.

Then we present in more detail the results for the offline (index-based) algorithms. We performed experiments with each algorithm and varied the underlying index structure, the parameters of the algorithm, the type of text, the text length n, the pattern length m, the distance measure δ, the search tolerance k, and the query type. Due to the size of this multidimensional space we cannot give all individual results but try to show some effects when varying the parameters independently and analyze the behavior of each algorithm in different settings. We therefore present detailed results for the two practically presumably most important types of text sequences in this context, namely natural language texts and DNA sequences. We furthermore chose the following parameter combination as pivot point for our experimental investigation:

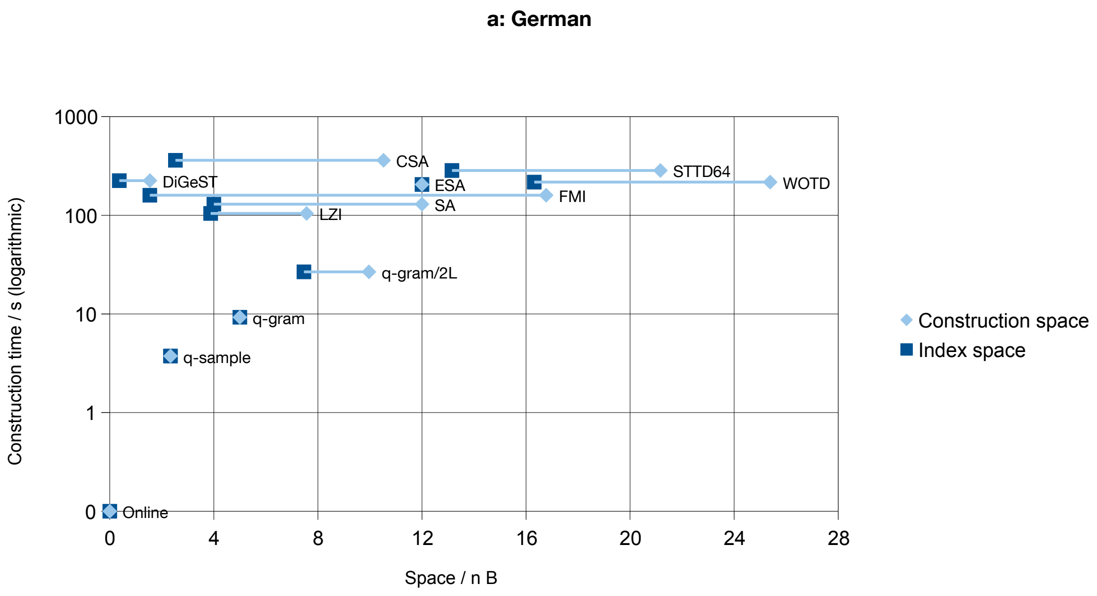
- Index: We use the enhanced suffix array because it is the default index structure and allows a suffix tree traversal.
- Text length: $n = 2^{26}$ is the largest text length where every index structure (except DiGeST) fits into main memory after it is completely constructed.
- Pattern length: $m = 16$ corresponds to (short sequences of) words or DNA fragments.
- Distance measure: $\delta = \delta_{\text{edit}}$ (simple edit distance) is a very popular distance measure suitable in many practical applications while still allowing computationally efficient solutions.
- Search tolerance: $k = 2$ because $k = 1$ is too simple for many applications and higher search tolerances do not work efficiently with all algorithms.
- Query type: R_{bool} because this is the most basic and a very popular query type.

This pivot point permits to observe many phenomena we encountered. The results in the following sections are given as diagrams and the shaded areas (marked with ‘‘(E)’’ for English texts and ‘‘(D)’’ for DNA sequences) correspond to the pivot point to simplify connecting the same data points in the different diagrams. All given search times are for answering 1000 queries.
In the diagrams we also indicate the error levels and the total number of matches *occ* (accumulated for all search queries) because especially the search times regarding different texts are not directly comparable and both values can have a significant impact.
For each *parameterized* algorithm we give preferably general and transferable recommendations for good parameter values.

Following this, we experimentally compare combinations of index structures and search algorithms and furthermore investigate the behavior in external memory.

6.4.1 Online algorithms

Online algorithms for approximate pattern matching (without index structures, Section 3.2) are not the focus of this theses. They are, however, used in offline algorithms for verifying candidate positions and therefore we investigate their performance here to determine the best algorithms for different settings. The results are summarized in experimental maps (Figure 47).
When searching *all matches* of a pattern (query type = R_{pos} or R_{count}), the online algorithms scale linearly in the text length. This is, however, not true for existence queries (query type = R_{bool}), because the search can be stopped as soon as a match was found; this can happen very close to



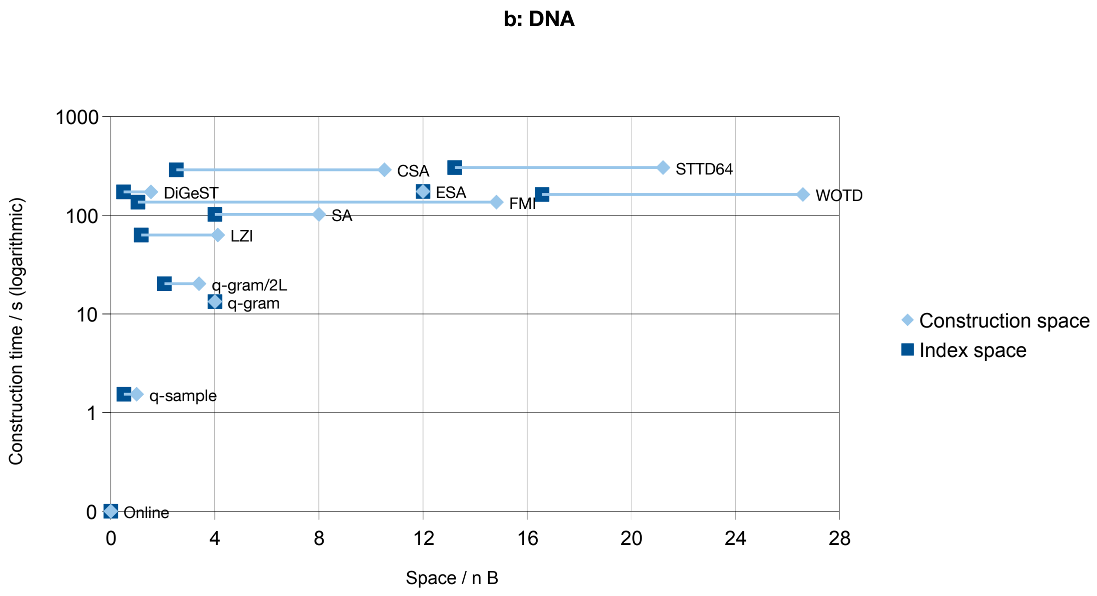


Figure 46: Index comparison: space usage and construction time.

(Using $n = 2^{26}$, t = `text-german`/`dna-human4`)

The diagrams show all implemented indexes with three important performance values of the construction:

- the construction time,
- the construction space (maximal main memory needed during the construction), and
- the index space (main memory usage of the fully constructed index).

We used the following parameters for the q-gram indexes: $q = 3$ for `text-german` and $q = 8$ for `dna-human4`. For the q-sample index we set *stepSize* = q. We otherwise use the parameters described in Section 6.3.

All index structures are here configured to reside in main memory, except for the DiGeST index which is a purely external memory index; its space usage is therefore not directly comparable here.

The underlying data of the diagrams can be found in the Appendix Section A.3.

(A similar diagram for space usage and search times of approximate pattern matching is given in Figure 54 on page 175.)

the beginning of the text, independently of the length (e. g., for very frequent patterns). Since the relative performance of the different online algorithms does not change much with growing text length we carry out the experiments for $n = 2^{20}$ here.

Dynamic programming The DP algorithm (`DPSearch`, Section 3.2.1) is not competitive to the bit-parallel variant by Myers (next paragraph). Only in cases where this algorithm cannot be used (e. g., when using a more complex string measure such as the weighted edit distance $\delta_{\text{edit, weighted}}$), we have to fall back to the general DP algorithm. In our experiments here we focus on the simple edit distance and therefore use the algorithm by Myers in the following.

Bit-parallel algorithm of Myers The algorithm of Myers (`Myers`, Section 3.2.2) turns out to be relatively independent of the type of text and of the chosen search tolerance k. It can be also used for high error levels $\alpha = \frac{k}{m}$. It runs faster for short patterns fitting into a machine word.

Splitting the pattern (PEX) The online partitioning algorithm (`Pex`, Section 3.2.3) is especially fast, if the pieces of the pattern are sufficiently long; in this case, a candidate match is with high probability also a real match. The length of the pieces corresponds to the reciprocal of the error level α and PEX is therefore fast if α is only moderate (e. g., $\alpha \leq 0.25$). This is the case if either the search tolerance k is low or the pattern is sufficiently long. However, for long patterns and boolean queries R_{bool}, the search time can even decrease with growing tolerance because matches can be found closer to the beginning of the text.
The hierarchical variant is never substantially slower than the non-hierarchical variant and we therefore use `TVerification = Hierarchical`. For finding the pieces, the algorithm by Wu and Manber [WM94] was fastest in all our experiments and we therefore use `TMultiFinder = WuManber`. In the following, the term *PEX* refers to this specialization.

Backward Automaton (ABNDM) The algorithm based on an automaton (`AbndmAlgo`, Section 3.2.4) is generally slower for longer patterns and higher search tolerances. Only in a few special cases it is the fastest among all algorithms.

To find the respective best algorithm for practically interesting settings, we performed a series of experiments with 4 different types of texts, different pattern lengths and tolerances. The results are presented graphically as *experimental maps* showing for each setting which algorithm is fastest (Figure 47); the type of diagram is based on [Jok+96; NR02; Aïc06]. We use the information of the experimental maps to choose the best verification algorithm for the index-based algorithms described below. (We do, however, not use the ABNDM algorithm since is is limited to a few special cases and the respective second best algorithm is never much slower.)

(Unfortunately none of the online algorithms was able to handle the wide character type `wchar_t` and failed with a segmentation fault when performing experiments with `text-chinese`.)

In the following sections we analyze the behavior of the *index-based* algorithms; we also include an *online* algorithm in the diagrams to illustrate its behavior as well. The relative performance of online and offline algorithms is, however, not directly comparable because it depends heavily on the query type and search tolerance etc.: for R_{bool} with a high search tolerance, a match can be expected to be found very close to the beginning of the text; when using R_{pos} or a low search tolerance, online algorithms might have to scan the whole text. The shown results are therefore not necessarily generalizable to other settings.

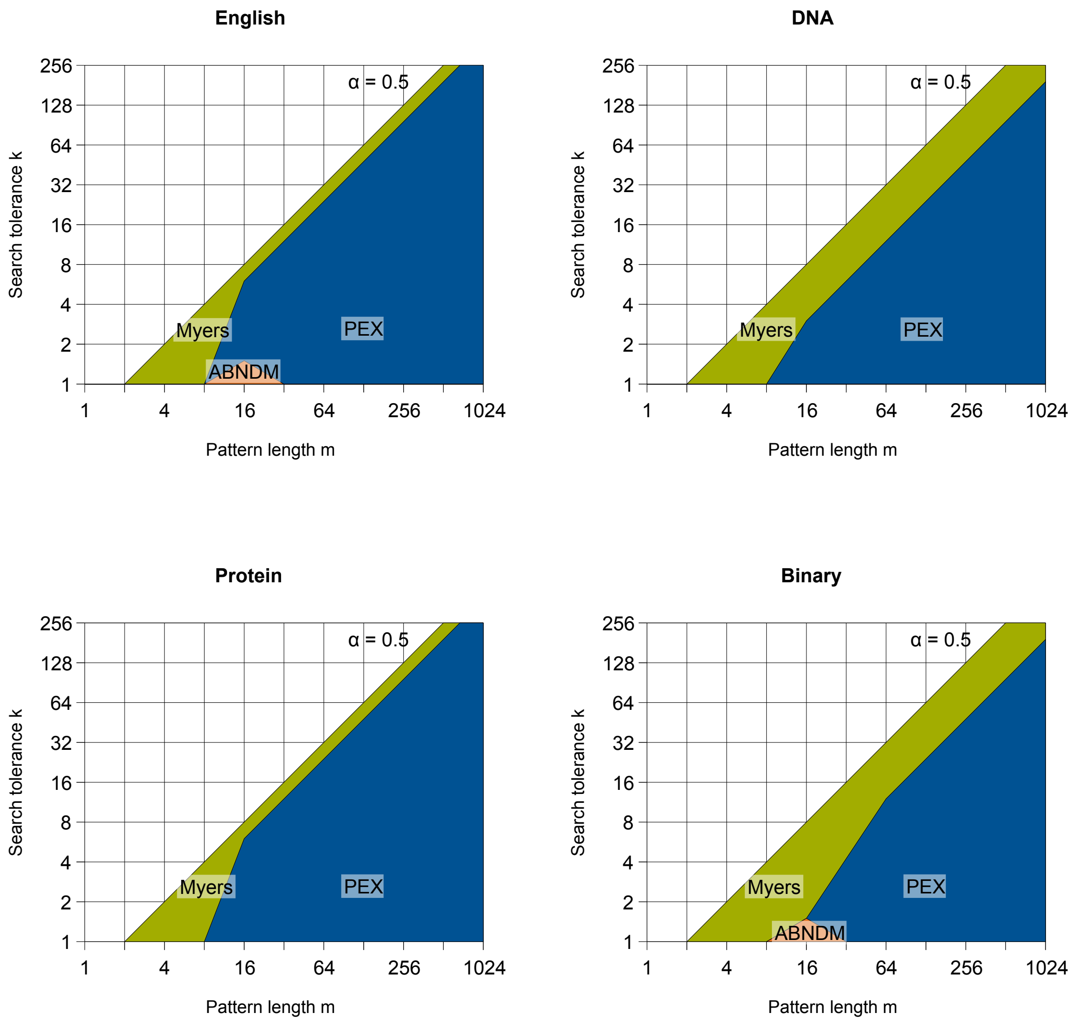


Figure 47: Online algorithms: experimental maps.

(Determined using $n = 2^{20}$, δ_{edit}, query type = R_{bool})

The experimental maps show for 4 representative types of texts which online algorithm performs best in which setting, depending on pattern length m and search tolerance k using an error level of at most $\alpha = 0.5$.

PEX is fastest in many cases, in particular for sufficiently long patterns and moderate error levels. The performance degrades for higher error levels because the filtering does not work efficiently in these cases ($\alpha > 0.25$ for big alphabets or $\alpha > 0.125$ for small alphabets).
The algorithm by Myers is in general fastest for short patterns and for high error levels where PEX is not efficiently usable. ABNDM is only fastest in very special cases with $k = 1$.

As texts we used `text-english`, `dna-human4`, `protein-all`, and `uniform-binary`. The experimental maps look very similar for other texts of the same alphabets (e. g., the synthetically generated counterparts).

6.4.2 Backtracking in tries and suffix trees

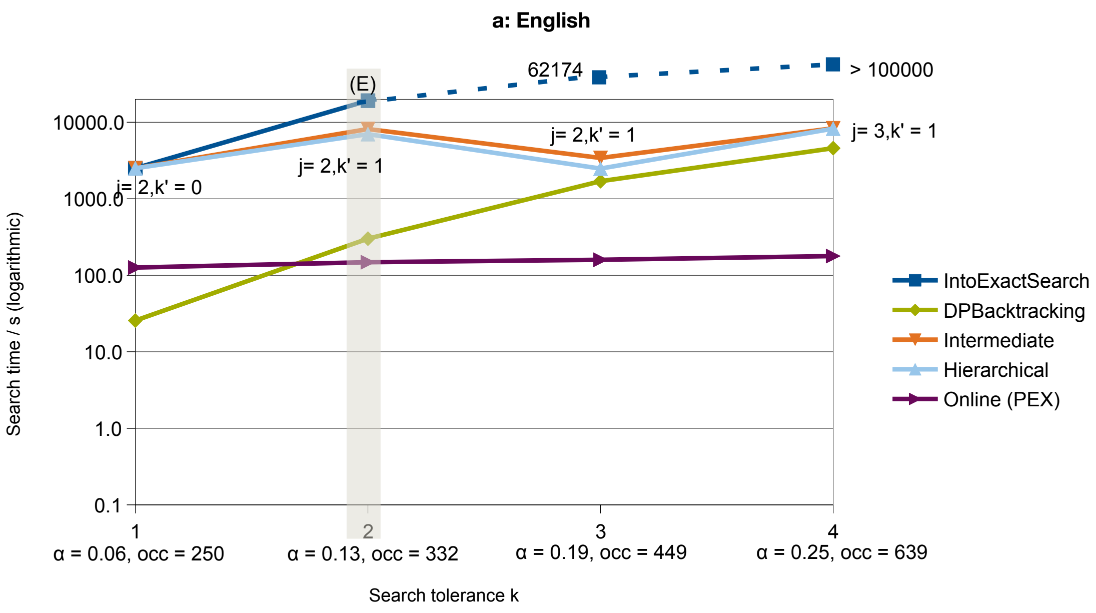
The dynamic programming algorithm for searching in tries performs backtracking in tree nodes (`DPBacktracking`, Section 3.3.2).
For increasing values of the search tolerance k, the theoretical worst case search time increases exponentially. In our experiments we observed that the dependence is sub-exponential in practice, e. g., for `text-english` (Figure 48a). However, backtracking is practically still infeasible for big alphabets and high search tolerances, especially in combination with long patterns.
We observed, however, that under certain conditions (small alphabets, query type = R_{bool}, short patterns), the search time does not increase beyond a certain point because the backtracking is stopped as soon as one match was found (for example with DNA sequences, Figure 48b).
Backtracking is furthermore extremely susceptible regarding the pattern length (Figure 49). It is very fast for short patterns because only a limited tree traversal up to depth $m + k$ has to be performed and no further verification is necessary. For long patterns, a great part of the suffix tree needs to be traversed and backtracking becomes infeasible. The turning point compared to the partitioning algorithm is $m \approx 70$ for natural language texts and $m \approx 30$ for DNA sequences (when using $k = 2$).

When analyzing the dependence on the characteristics of the underlying text, we found that neither the alphabet size nor the inverse probability of matching or the distribution of q-grams serve as a good predictor for the search time. It turned out that we additionally have to take into account the correlations of the text symbols and there seems to be a roughly exponential connection between the empirical entropy H_0 of order 0 and the search time (Figure 51). The order of the test instances with respect to H_0 is in particular the same as regarding the search time (the only exception is `uniform-protein`).
The backtracking algorithm usually has a sublinear dependency of the search time on the text length (Figure 50a), because in most settings the time is dominated by the tree traversal – and not by outputting the matches of the found subtrees. We observed that under certain conditions (small alphabets, query type = R_{bool}, short patterns), the search time even *decreases* with longer texts (Figure 50b): this is because a longer text leads to a denser suffix tree and matches can potentially be found earlier.

Using backtracking with a suffix tree which does not fit into main memory but is stored in external memory is not efficiently possible because each traversal in the tree might lead to a slow I/O operation. In our experiments with DNA sequences, the enhanced suffix array for $n = 2^{26}$ fits into main memory and the search time is 2.6 s; it does not fit for $n = 2^{28}$ ($S_{\text{ESA}} = 12\,n\,\text{B} = 3.1\,\text{GiB} > 1.4\,\text{GiB}$ of main memory) which results in a much higher search time of 546.7 s (not presented graphically).

6.4.3 Partitioning into exact search

The partitioning algorithm based on the pigeonhole principle in its basic form (`Partitioning<IntoExactSearch>`, Section 3.3.3) has a few parameters. For searching the pieces we rely on the standard exact pattern matching algorithm provided by the index structure. For verifying candidate positions we choose an online algorithm based on our experimental maps (Figure 47). (In the experiments we noticed that the algorithm by Myers is in some cases favorable over PEX deviating form the experimental maps, presumably because the verifications are only carried out in very small regions of the text.)
For the number of pieces we observed that the overall search is fastest if the pattern is split into as few pieces as possible by using $j = k + 1$ (which is also the default setting) to obtain a good filtering effect.



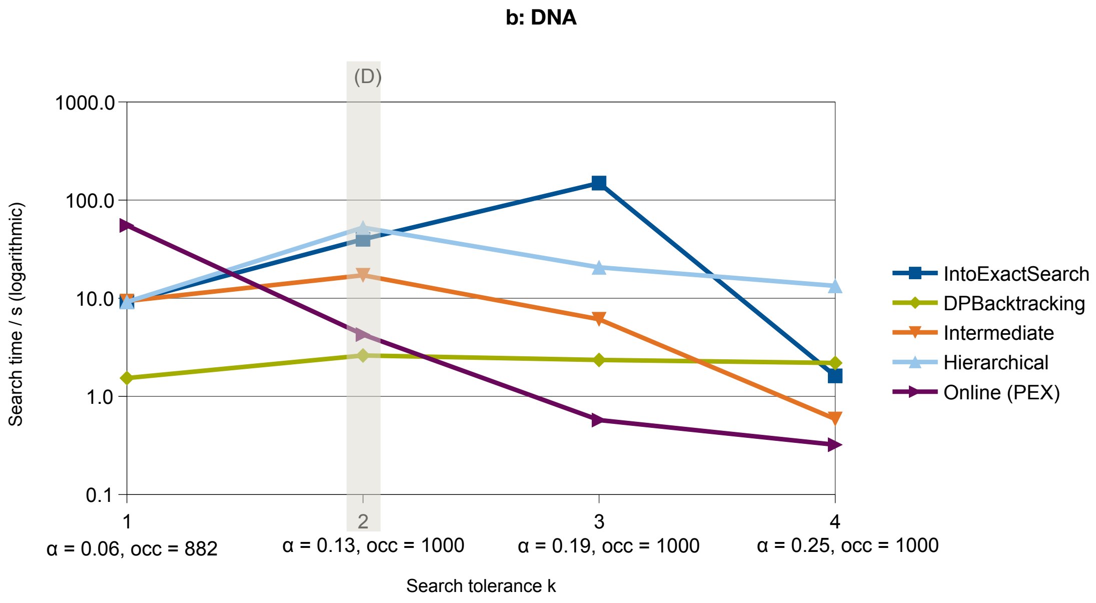


Figure 48: Index algorithms: dependence on search tolerance k.

(Using t = `text-english` and `dna-human4`, $n = 2^{26}$, $m = 16$, δ_{edit}, query type = R_{bool})
The pivot point of our experiments is indicated by (E) and (D).
The behavior of the algorithms is described in more detail in the corresponding sections.

Just as the backtracking algorithm (see above), also the partitioning algorithms performs worse for high search tolerances k. However, while backtracking is generally slow for high *absolute* values of k, partitioning is generally slow for high *relative* values $\alpha = \frac{k}{m}$. This is the case because a high error level leads to small pieces which tend to occur often in the text yielding many candidate positions and a slow search (Figure 48a). We observed, however, that under certain conditions (small alphabets, query type = R_{bool}, short patterns) an increased search tolerance can lead to a faster search because most patterns are contained in the text and among the first few candidate positions is a match (Figure 48b); note that if a pattern does *not* have a match, all candidates need to be checked.
The search time decreases with increasing pattern length (Figure 49) because of the decreasing error level and the resulting longer pieces with better filtering. Only for very long patterns (in our experiments for $m = 1024$) it increases again due to the verification costs which grow with the pattern length.

The partitioning algorithm performs quite differently on text with different characteristics. It is generally faster for larger alphabets if the characters are uniformly distributed (Figure 52). If the text has a skewed character distribution or a higher order structure, this is, however, not necessarily true. When trying to describe the search times based on statistical measures for the character distribution of the text, we found that neither the total alphabet size nor the actually used alphabet size, the inverse probability of matching, or the empirical entropy H_0 of order 0 are suitable measures because they are not able to describe the observed variance. It turned out, however, that higher order entropies (such as H_g for $g = 5$ which we computed for all test instances) can better describe the variance: a text with high entropy H_g needs significantly less time than a text with low entropy because the implied randomness of the text yields a more effective filtering (Figure 52).
The dependence of the search time on the text length is roughly linear (Figure 50) because the number of candidate positions grows linearly in the text length.

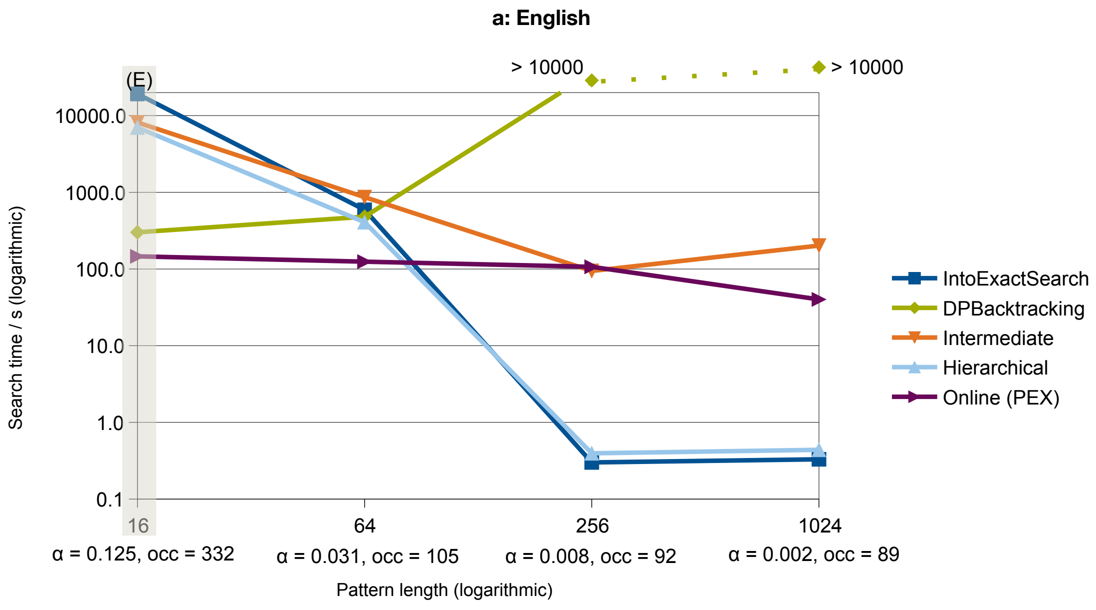
Just as the backtracking algorithm, also the partitioning algorithm does not work well in an external memory setting because each search for a piece might lead to a costly I/O operation. In our experiments with DNA sequences, the enhanced suffix array for $n = 2^{26}$ fits into main memory and the search time is 39.8 s; it does not fit for $n = 2^{28}$ which results in a much higher search time of 2342.3 s (not presented graphically).

Recommendation for algorithm parameters of `Partitioning<IntoExactSearch>`:

- `TPieceFinderSpec = Default` and `TPiecePatternSpec = Default`.
- `TVerifyFinderSpec` and `TVerifyPatternSpec`: Choose based on the experimental maps (Figure 47).
- `setNumberOfPieces(`$k+1$`)`.

6.4.4 Intermediate partitioning

The intermediate partitioning algorithm (`Partitioning<Intermediate>`, Section 3.3.4) permits a trade-off between both algorithms. The parameter j controls the number of pieces and allows to choose between the pure backtracking algorithm ($j = 1, k' = k$) and partitioning into exact search ($j = k + 1, k' = 0$) where k' denotes the tolerance used to search the pieces. Choosing an appropriate value for the parameter j is crucial to achieve a good performance, we observed that the search times vary by several orders of magnitudes.



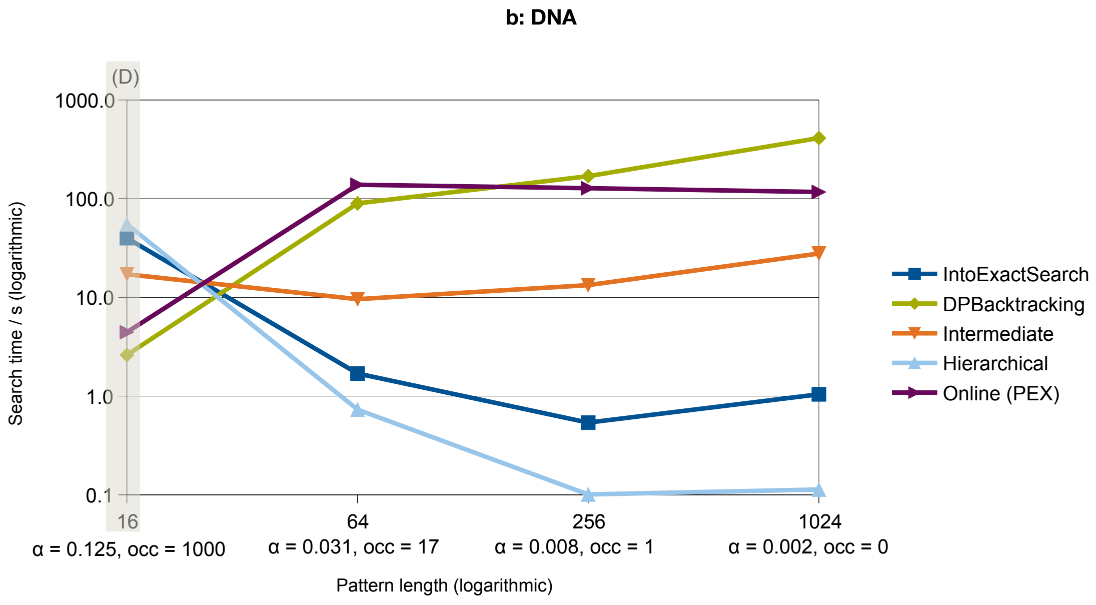


Figure 49: Index algorithms: dependence on pattern length *m*.

(Using t = `text-english` and `dna-human4`, $n = 2^{26}$, δ_{edit}, $k = 2$, query type = R_{bool})
The behavior of the algorithms is described in more detail in the corresponding sections.

$k =$					
256					99 (3)
128					89 (1)
64				25 (3)	84 (1)
32				23 (1)	82 (0)
16			7 (2)	21 (1)	80 (0)
8			6 (1)	21 (0)	80 (0)
4		2 (2)	6 (1)	20 (0)	80 (0)
2		2 (1)	6 (0)	20 (0)	79 (0)
1	1 (1)	2 (1)	5 (0)	20 (0)	79 (0)
j^* (k')	**4**	**16**	**64**	**256**	**1024** $= m$

$k =$					
256					65 (3)
128					65 (1)
64				17 (3)	33 (1)
32				17 (1)	33 (0)
16			6 (2)	9 (1)	17 (0)
8			5 (1)	9 (0)	9 (0)
4		2 (2)	3 (1)	5 (0)	5 (0)
2		2 (1)	3 (0)	3 (0)	3 (0)
1	1 (1)	1 (1)	2 (0)	2 (0)	2 (0)
j^{**} (k')	**4**	**16**	**64**	**256**	**1024** $= m$

Table 12: Index algorithms: optimal choice of parameter j for intermediate partitioning.

(Exemplary for t = dna-human4, $n = 2^{26}$, $\delta = \delta_{\text{edit}}$, query type = R_{bool}, $\alpha \leq 0.25$, enhanced suffix array.)

The tables show two strategies for choosing a good value of the parameter j of the intermediate partitioning algorithm given the pattern length m and the search tolerance k. Each cell contains the proposed choice for the number of pieces j (together with the corresponding tolerance k' for searching pieces). The case $j = 1$ (and $k' = k$) corresponds to pure backtracking, the case $j = k + 1$ (and $k' = 0$) corresponds to partitioning into exact search.
The left table uses the formula for j^* of Navarro and Baeza-Yates [NBY00], the right table our adjusted formula for j^{**} which employs the same piece tolerance but longer pieces.
In our experiments, our adapted value j^{**} was favorable over j^* and the optimal choice in nearly all tested cases of DNA sequences (except for $m = 16$, $k = 2$ and $m = 64$, $k = 4$ where the experimentally optimal j was 1 and 5, respectively). The results are similar for other types of text.

A detailed analysis of the parameter choice is given by Navarro and Baeza-Yates [NBY00]; for practical cases they propose to use the following value j^* depending on the pattern length m, the search tolerance k, the alphabet size σ, and the text length n:

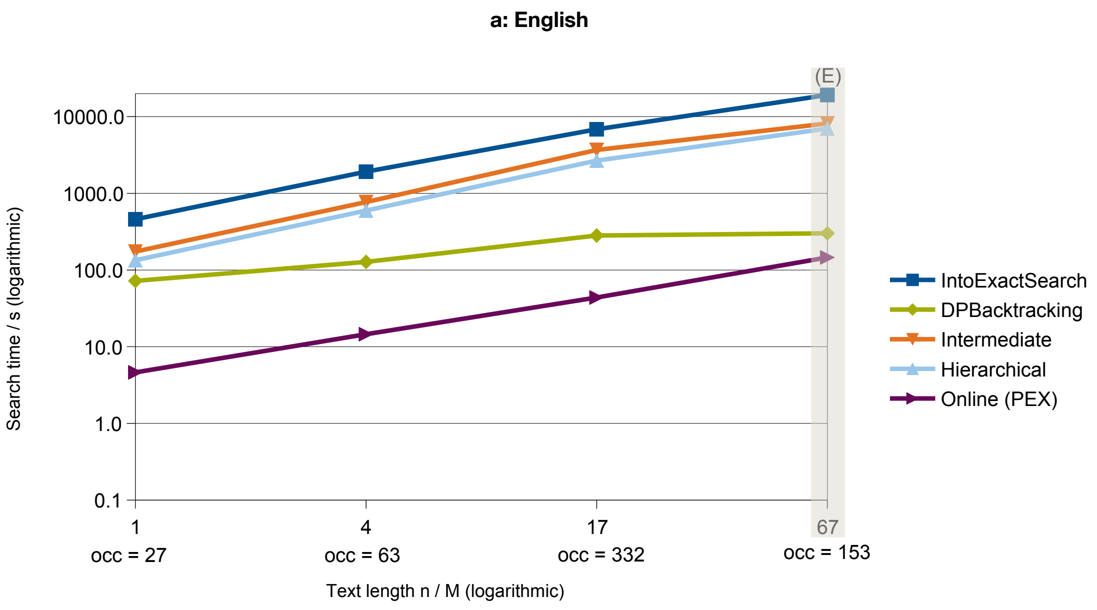
$$j^* = \frac{m + k}{\log_\sigma n}$$

As a measure for the alphabet size, however, we do not use the total size $|\Sigma|$ of the underlying alphabet Σ but the inverse probability of matching (which equals the alphabet size for uniformly distributed texts but is, e. g., ≈ 15 for natural language texts, see Table 6 on page 119). This is suggested to give better results for skewed character distributions [NBY00].
We performed a series of experiments with different types of text (English, DNA, protein, and binary) and different values for m and k. For each setting we determined the experimentally optimal value of j. It turned out that the theoretical formula often yields too high a value j^*, i. e., too many pieces which are in turn too short and have many candidate matches resulting in a poor performance (the intermediate partitioning algorithm turns out to be very susceptible regarding the choice of j). However, we noted that when calculating the *piece tolerance* (= tolerance for searching the pieces of the pattern) as $k' = \left[\frac{k}{j^*}\right]$, it is predicted optimally by the theoretical formula in most cases.[9] We therefore adapted the formula to use the *smallest* number j^{**} of pieces having the *same predicted piece tolerance* as j^* as follows:

$$j^{**} = \left\lfloor \frac{k}{k' + 1} \right\rfloor + 1 = \left\lfloor \frac{k}{[k/j^*] + 1} \right\rfloor + 1$$

This results in longer pieces which in turn achieve a better filtering effect and therefore shorter search time. Table 12 shows the theoretically recommended values j^* and our adjusted values j^{**} for DNA sequences. For $m = 64$, $k = 4$, e. g., the theoretical value $j^* = 6$ gives a piece length $m' \approx 11$ and our adjusted value $j^{**} = 3$ gives a piece length $m' \approx 21$ with a much better filtering.

[9]The brackets denote commercial rounding here: $[x] := \left\lfloor x + \frac{1}{2} \right\rfloor$.



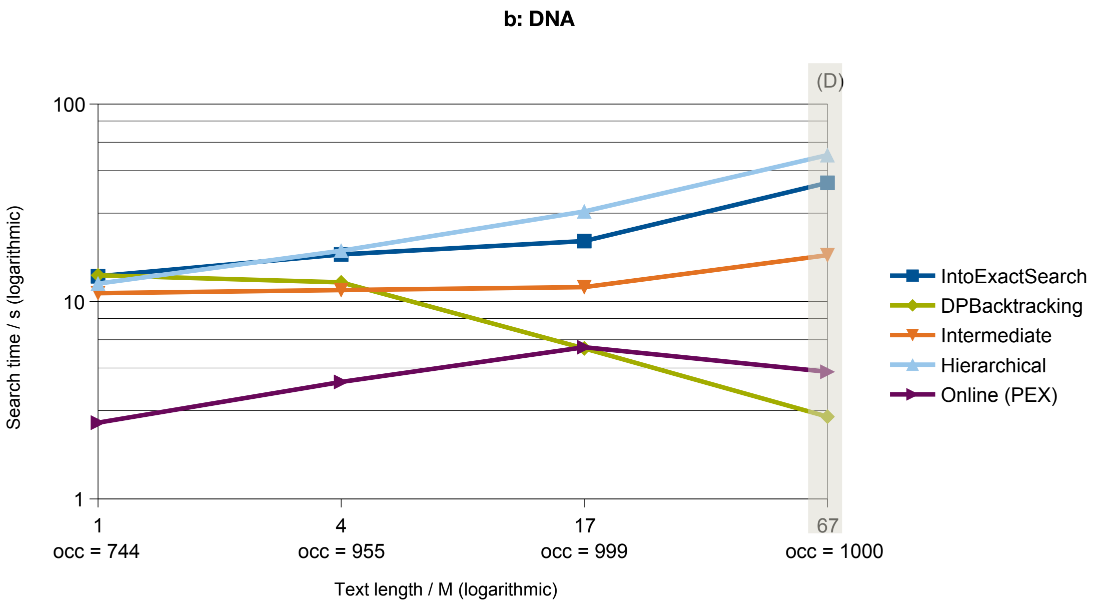


Figure 50: Index algorithms: dependence on text length *n*.

(Using t = `text-english` and `dna-human4`, $m = 16$, δ_{edit}, $k = 2$, query type = R_{bool})

The behavior of the algorithms is described in more detail in the corresponding sections.

It turned out that our adjusted computation equals the experimentally observed optimal value in nearly all settings of DNA, protein, and binary sequences (only in a very few cases, a slightly higher j and therefore lower k' is be preferable). For natural language texts, however, j^{**} is systematically predicted higher than the experimentally observed optimum; we fix this by using an even smaller value for the alphabet size to accommodate for the skewed character distribution.

To investigate the dependence of the search time on the tolerance (Figure 48), the pattern length (Figure 49), and the text length (Figure 50) we use the respective experimentally optimal j in each setting (but excluded the extreme cases backtracking $j = 1$ and partitioning into exact search $j = k + 1$ to get a real intermediate case). It turns out that intermediate partitioning is faster than the extreme cases for sufficiently long patterns and high error levels α. We performed series of experiments with different pattern lengths $m \in \{4, 16, 64, 256, 1024\}$ and search tolerances $k \leq 256$ and determined this turning point; approximate values are for natural language texts $\alpha \geq \frac{1}{16}$, for DNA sequences $\alpha \geq \frac{1}{8}$, for protein sequences $\alpha \geq \frac{1}{4}$, and for binary sequences $\alpha \geq \frac{1}{16}$. If we also allow the extreme cases for choosing j, the performance equals the best of the three algorithmic variants in each setting (with a small slow-down due to the overhead of j being not fixed at compile time).

Recommendation for algorithm parameters of `Partitioning<Intermediate>`:

- `TPieceFinderSpec=DPBacktracking`, `TPiecePatternSpec=DPBacktracking`.
- `TVerifyFinderSpec` and `TVerifyPatternSpec`: Choose based on the experimental maps (Figure 47).
- `setNumberOfPieces(`j^{**}`)` applying our adjusted computation based on [NBY00] and using the inverse probability of matching as measure for the alphabet size.

6.4.5 Partitioning with hierarchical verification

Partitioning with hierarchical verification (`Partitioning<PartitioningHierarchical>`, Section 3.3.5) splits the pattern recursively into pieces, forming conceptually a tree of pieces. The branching width of this tree can be controlled with the number j of pieces in each level. (The height h of the tree can be controlled by nesting h hierarchical finder classes and using partitioning into exact search on the bottom level.)

In our experimental series with different types of texts, pattern lengths, and tolerances, we found that in most scenarios it is fastest to choose $j \in \{2, 3\}$. To fully exploit the benefits of the hierarchical verification it should hold for the height h of the tree: $k + 1 = j^h \Leftrightarrow h = \log_j(k + 1)$. In most cases, however, the hierarchical variant is already for $h = 2$ faster than the basic partitioning algorithm. The dependence of the performance of the hierarchical algorithm on j and h is fortunately not as fragile as the dependence of the *intermediate* partitioning algorithm on j.
The algorithm performing the verifications should be chosen according to the experimental map for online algorithms (Figure 47). (Note, however, that this choice has to be made based on the tolerance and the piece length *in each level* and not for the whole search pattern.)

The hierarchical verification algorithm is orders of magnitudes faster than the basic partitioning algorithm for higher search tolerances and error levels because most of the costly verifications can be skipped. The trade-off between the algorithms is summarized for four different representative types of text in the experimental map in Figure 53.

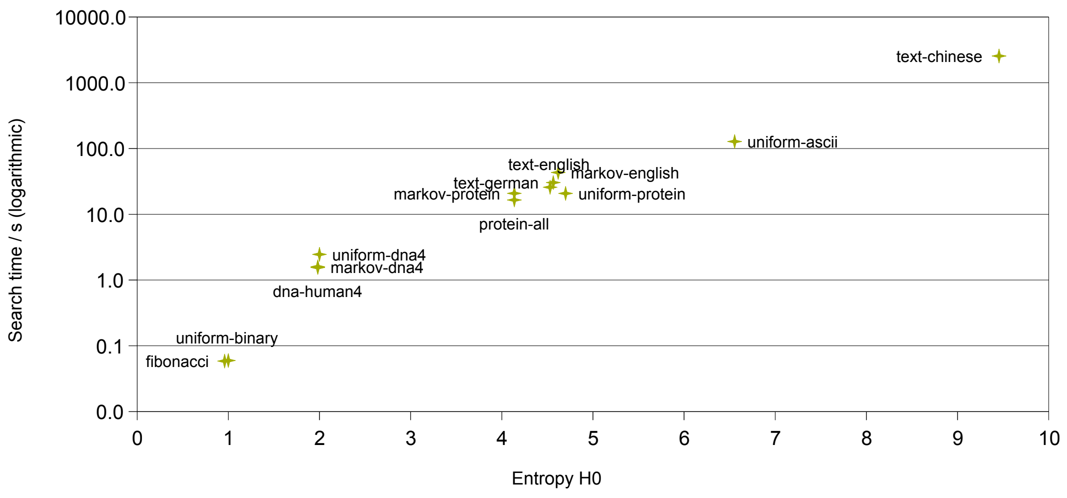


Figure 51: Index algorithms: dependence on entropy (backtracking algorithm).

(Using $n = 2^{26}$, $m = 16$, δ_{edit}, $k = 1$, query type = R_{bool})

The search time of the backtracking algorithm grows with increasing empirical entropy H_0 of the underlying text. There is no such strong relation for other statistical measures of the text, such as the alphabet size, the inverse probability of matching, number of different q-grams etc.

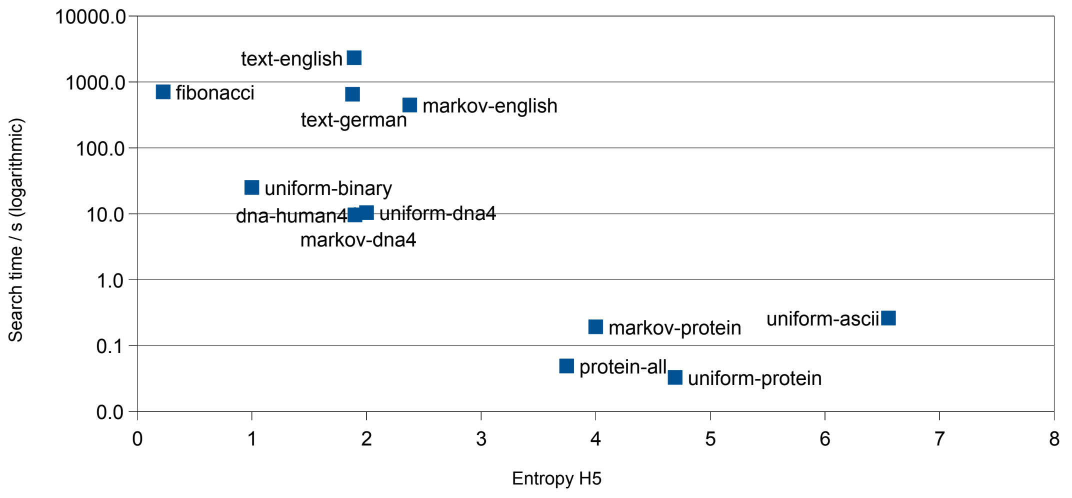


Figure 52: Index algorithms: dependence on entropy (partitioning algorithm).

(Using $n = 2^{26}$, $m = 16$, δ_{edit}, $k = 1$, query type = R_{bool})

The search time of the partitioning algorithm is negatively correlated to the empirical entropy H_5 of the underlying text. There is no such strong relation for other statistical measures of the text, such as the alphabet size, the inverse probability of matching, number of different q-grams etc.
(The instance `text-chinese` is not included because the online verification algorithms do not work for wide characters.)

The behavior regarding search tolerance (Figure 48), pattern length (Figure 49), and text length (Figure 50) is otherwise similar to partitioning into exact search; hierarchical verification is furthermore never significantly slower.

Recommendation for parameters of `Partitioning<PartitioningHierarchical>`:

- `TPieceFinderSpec = FinderPartitioning` and `TPiecePatternSpec = Partitioning`. (The height *h* of the verification tree can be controlled by nesting $h - 1$ classes `PartitioningHierarchical` and one class `IntoExactSearch` at compile time.)
- `TVerifyFinderSpec` and `TVerifyPatternSpec`: Choose based on the experimental maps (Figure 47).
- `setNumberOfPieces(j)` with $j \in \{2, 3\}$ for most practical settings.

6.5 Combinations of index structures and search algorithms

In the last sections we investigated the behavior of index structures on the one hand and approximate search algorithms on the other hand; we treated both parts relatively independently so far. In this section we investigate the behavior of *combinations of index structures and algorithms*. We focus especially on the *partitioning into exact search* algorithm because it can be used with all indexes and is the basis for the intermediate partitioning algorithm and for partitioning with hierarchical verification.
For index structures and algorithms *with parameters* we use the above determined and indicated optimal / practically suitable values in each setting. It is important to note, however, that the some parameters influence the performance and allow a trade-off between space and time needed. We first investigate the behavior in *main memory* – the behavior in *external memory* is discussed in the following section.

Backtracking. The backtracking algorithm can be used with the enhanced suffix array and the WOTD suffix tree. The performance of both index structures is very similar, but WOTD (eager version) is generally slightly faster (up to 20 %) because the enhanced suffix array has to perform some computations to traverse the tree. Searches in the *lazy version* of the WOTD suffix tree take longer time because the tree structure has to be built during the traversal; we do not investigate the behavior of the lazy variant in more detail here because the state of the tree would depend on the *history* of previous searches.

Partitioning into exact search. To compare the all index structures regarding the space usage and search time, we present them exemplarily in four diagrams (Figure 54): for natural language (German) texts and for DNA sequences and two different pattern lengths $m \in \{64, 1024\}$ using $k = 2$. This hopefully makes it easy to select an index structure based on the available memory and requirements for the time to answer search queries. Note, however, that the diagram does not show all aspects since some index structures offer extended functionality or can better be used in external memory. In the following, we briefly describe some findings.

The classical suffix array (SA) is smaller than the enhanced suffix array (ESA); it is, however, therefore also slower at answering search queries.
The WOTD suffix tree is slower and bigger than ESA when using the partitioning algorithm (while WOTD was faster when using the *backtracking* algorithm).

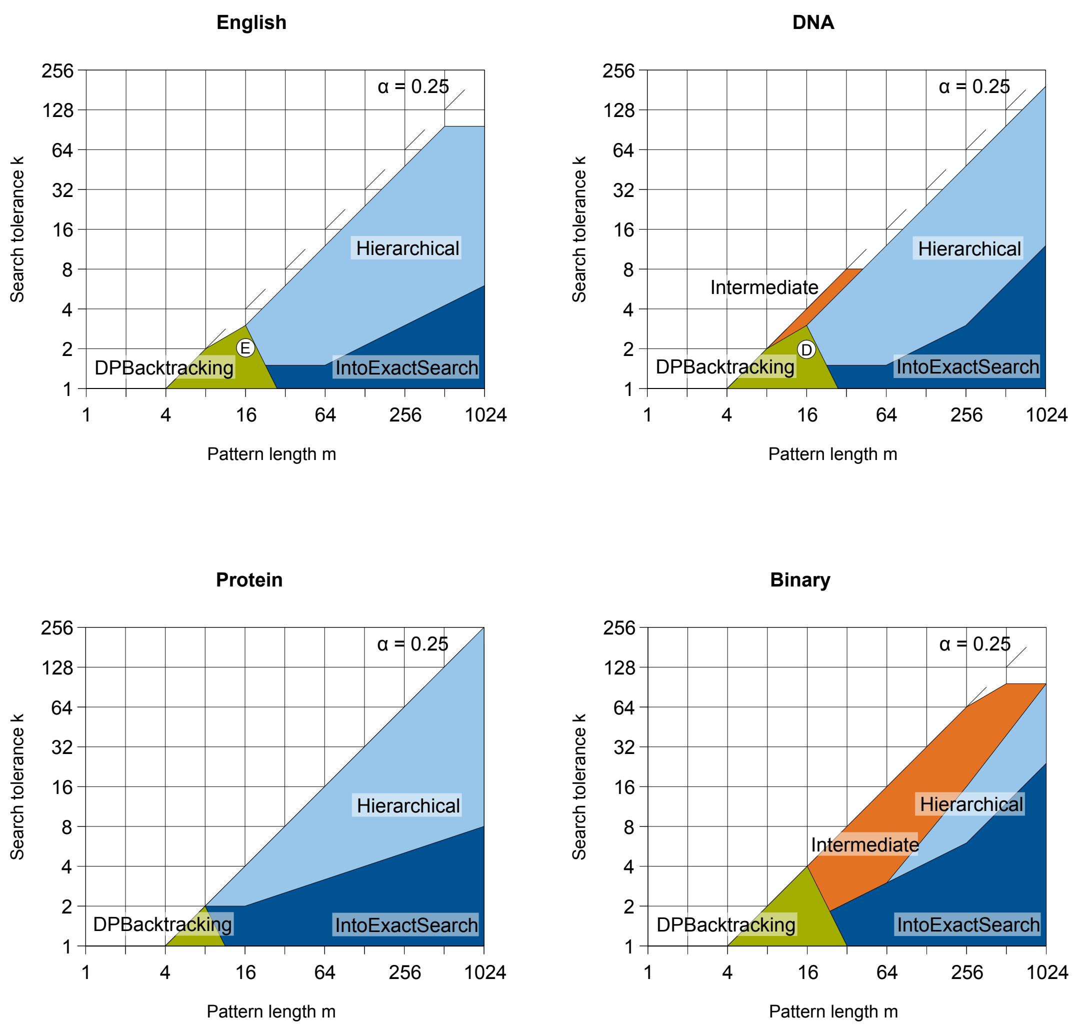


Figure 53: Index algorithms: experimental maps.

(Determined using $n = 2^{26}$, enhanced suffix array, δ_{edit}, query type = R_{bool})

The experimental maps show for 4 representative types of texts which offline algorithm performs best in which setting depending on pattern length m and search tolerance k using an error level of at most $\alpha = 0.25$.

For short patterns, the backtracking algorithm is generally best, while partitioning into exact search is best for longer patterns and low tolerances, and partitioning with hierarchical verification is best for longer patterns and high tolerances. Intermediate partitioning is only in a few settings with small alphabets the best algorithm.

As texts we used `text-english`, `dna-human4`, `protein-all`, and `uniform-binary`. The experimental maps look very similar for other texts of the same alphabets (e. g., the synthetically generated counterparts).
The experimental map is left empty for some settings with high error levels $\alpha = \frac{k}{m}$ (e. g., $\alpha = \frac{1}{4}$) where no algorithm was able to answer the 1000 queries within 1 h.

The map looks very similar for the q-gram index. We did not perform all experiments with all index structures, but at least the general behavior of the algorithms can be assumed to be similar. For indexes that do not support a suffix tree traversal, the choice is limited to partitioning into exact search and partitioning with hierarchical verification.

The STTD64 representation is significantly faster than WOTD and ESA for natural language texts because it can make use of the string depth stored in the leaves. STTD64 is slower than WOTD for DNA sequences with query type R_{bool} because our implementation searches all matches of a piece initially, instead of searching them one after the other as done by WOTD. This negatively influences the performance when searching for pieces with many matches (as occurring when searching for short patterns with small alphabets). This is, however, only a property of our implementation and cannot be attributed to the data structure itself. We plan to change this aspect in a future version of the implementation. When searching *all matches* (query type R_{pos}), our implementation of STTD64 is again faster than WOTD (not shown graphically here).
The DiGeST index is relatively slow compared to most other index structures in cases where the indexes fit into main memory, since DiGeST is especially optimized for external memory. The performance in an external memory setting is discussed in the following section.

The compressed index structures allow a trade-off between a faster search and a smaller index by using the respective parameters for the sample rates. When choosing a lower sample rate, the index is bigger, but faster; when choosing a higher sample rate, the index is smaller but slower. Here we use the parameters described in Section 6.3.
The CSA is smaller than the classical suffix array but therefore slower at answering search queries. It is not competitive for longer search patterns.
The FMI is smaller than most other indexes while it is still able to answer approximate search queries relatively fast. It is particular faster than the other compressed indexes in most cases.
The LZI is in comparison generally very small, especially for natural language texts. It is, however, not competitive with respect to the performance of answering search queries.

Regarding space and search time, the classical q-gram index behaves relatively similar to the suffix array (because both use the same main data structure); here we use $q = 3$ for natural language texts, and $q = 8$ for DNA sequences.
The q-sample index (here with *stepSize* := q) is significantly smaller than the q-gram index, while still being similarly fast. The theoretical slowdown with factor *stepSize* is not very remarkable in practice.
The q-gram/2L index has no advantage when using natural language texts. It is, however, smaller than the classical q-gram index when using DNA sequences and even faster in some settings (here for $m = 16$): The classical q-gram index has to verify the whole search pattern as soon as a q-gram matches; the q-gram/2L index first performs a comparison of the corresponding subsequence and immediately continues with the next subsequence in case of inequality (see Section 2.4.3).

Intermediate partitioning. The search time of the intermediate partitioning algorithm is very similar for WOTD and ESA. While ESA had an advantage for partitioning into exact search and WOTD had an advantage for backtracking, both index structures have nearly exactly the same performance for the intermediate partitioning algorithm in all tested settings with different pattern lengths and search tolerances.

Partitioning with hierarchical verification. The relative performance of the index structures for partitioning with hierarchical verification is very similar to partitioning into exact search. This is not surprising since the only difference is in the verification procedure which is independent of the underlying index structure. We therefore do not give detailed results here.

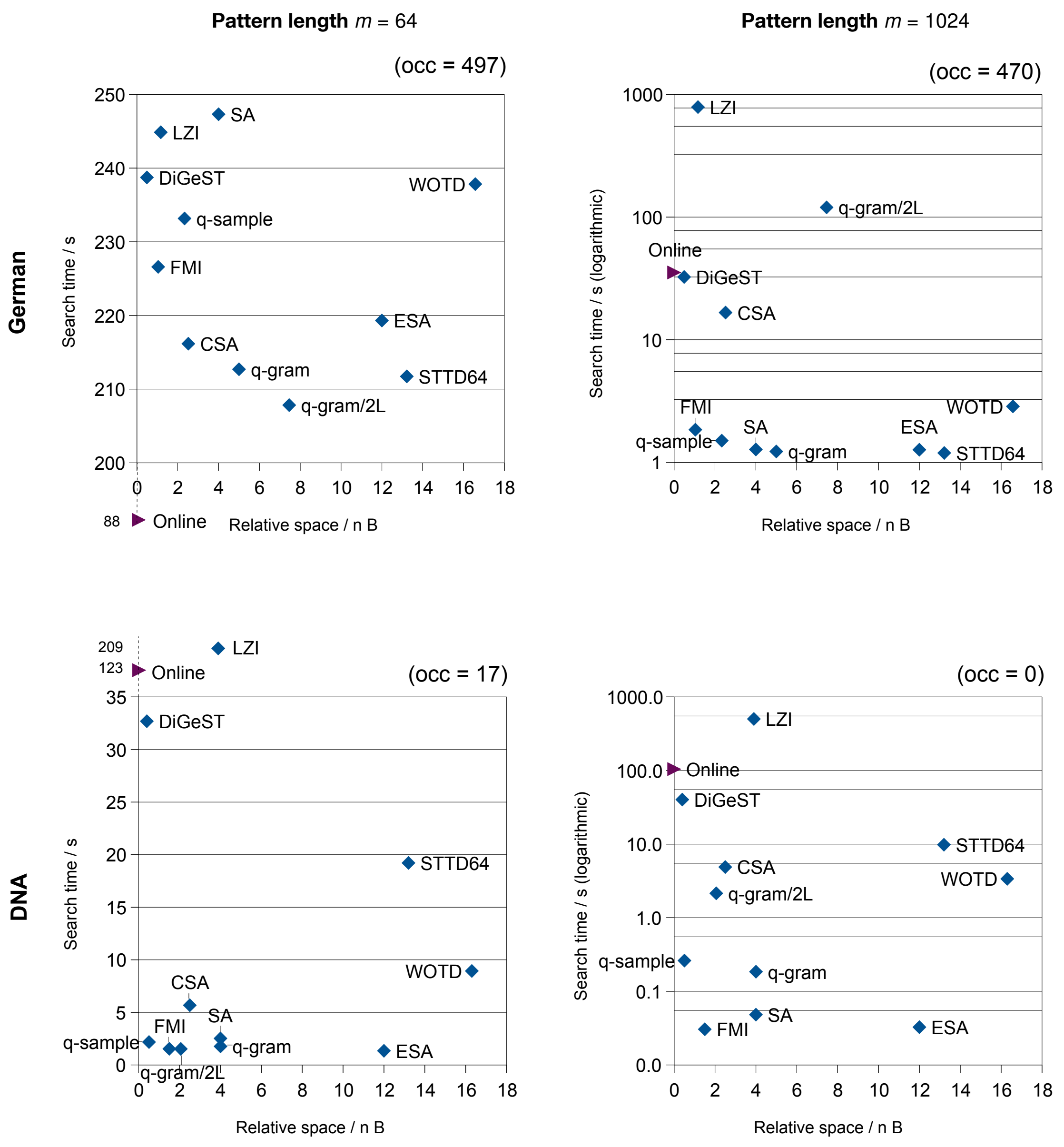


Figure 54: Index comparison: space usage and search time.

(Using $n = 2^{26}$, t = `text-german` / `dna-human4`, $\delta = \delta_{\text{edit}}$, $k = 2$, R_{bool})

The diagrams show all implemented indexes with two important performance values for approximate searching using partitioning into exact search:

- the index space (space usage of the fully constructed index).
- the search time (for answering 1000 queries)

The results are given for four representative combinations of text types (natural language text and DNA sequences) and pattern lengths (64 and 1024).
We used the following parameters for the q-gram indexes: $q = 3$ for `text-german` and $q = 8$ for `dna-human4`. For the q-sample index we set *stepSize* = q. We otherwise use the parameters described in Section 6.3.

The search times for `text-german` with $m = 16$ are dominated by the verifications; the relative differences between the indexes are therefore only small.

All index structures are configured to reside in main memory, except for the DiGeST index which is a purely external memory index; its space usage is therefore not directly comparable here.

The performance of an online algorithm (here: PEX) is shown as well for comparison.

The underlying data of the diagrams can be found in the Appendix Section A.3.

(A similar diagram for the construction space and construction time is given in Figure 46 on page 161.)

6.6 Approximate search in external memory

The behavior of the index structures and algorithms is substantially different when used in external memory instead of main memory. Many indexes can practically not be constructed (as discussed above in Section 6.3) and also the search algorithms suffer from slow external memory access. We therefore investigate the practical usability of the implemented solutions for texts of size 1 GiB.

Suffix arrays and suffix trees. Neither the classical suffix array, nor the enhanced suffix array or the WOTD suffix tree (in the eager, as well as in the lazy version) can be used in such a setting. The text already occupies 1 GiB of main memory leaving only ≈ 350 MiB for the index construction after subtracting space for the operating system etc. (while each index is of size at least 4 GiB). Even if it possible to construct such an index (e. g., on a different computer with more main memory), approximate pattern matching algorithms like backtracking are not expected to work well due to many random memory accesses and I/O operations (as discussed in Section 2.2.7 and [Bar+11a]).
The STTD64 suffix tree can be built for bigger texts than WOTD by using the partitioning approach. Performing approximate pattern matching using the partitioning algorithm is, however, still slow because each search of a piece can require several I/O operations. We were able to build the STTD64 index for `uniform-ascii` of size 1 GiB = 2^{30} (this type of text is advantageous for STTD64 because it results in a comparably small index). Searching patterns of different length $m \in \{64, 256, 1024\}$ and different tolerances $k \in \{1, 2\}$ takes between 2000 s and 3000 s which is very slow compared to the DiGeST index investigated below.

Compressed indexes. None of the implementations of the compressed indexes offers an algorithm for the efficient construction in external memory. Neither the FMI not the CSA or the LZI can therefore be used for texts of length 2^{30} in our experiments.

***q*-gram indexes.** The q-gram index can efficiently be built in external memory; therefore, the parameter for the page size has to be given a sufficiently small value so that $\sigma^q + 1$ frames can be held in main memory for the *positions* table as discussed above (Section 6.3.10). We built the q-gram index for `dna-human4` using different values of q and different page sizes. We found that searches (here with $m = 64, k = 1$) are fastest for $q = 5$ or $q = 6$ using the highest possible page size determined as above for the construction (giving search times between 400 s and 500 s).
The q-sample index offers the possibility to reduce the space consumption by using a higher step size. We tried different combinations for q between 4 and 8 and *stepSize* between 2 and 16. We found that searches (here with $m = 64, k = 1$) are fastest for $q = 8$ and $\textit{stepSize} = 12$, presumably because the resulting index just fits into main memory (the positions table takes $4\,n\,\mathrm{B} / \textit{stepSize} = 341$ MiB, the directory is negligible). The search time is then approximately 600 s. We want to note, however, that even though this is an index structure working for 1 GiB texts on our computer, it cannot arbitrarily be extended to longer texts and is not comparable to indexes in external memory.

Suffix forests. Our experiments with the DiGeST suffix forest revealed that this index structure (which is especially designed for the use in external memory) is fastest among all indexes when searching in huge texts. The search time for `dna-human4` (of length $n = 2^{30}$ using $m \in \{64, 256, 1024\}, k = 1$) is between 40 s and 60 s and thereby an order of magnitude faster than the other implemented indexes. The parameter `OUTBUF_SIZE` controls the size of the partial trees and the experimentally optimal value for the *construction* (Section 6.3.6) is also optimal for *searching*.

	[Min+14]	[this work]
Approach	Backtracking	Partitioning
Distance measure	δ_{Hamming}	δ_{edit}
Text length n	95 MiB	64 MiB
Pattern length m	13	16
Processor	Intel Core 2 Duo	AMD Athlon XP 3000+
	2 cores at 2800 MHz	1 core at 2154 MHz
RAM	4000 MiB	1475 MiB

	Time	(*occ*)	Time	(*occ*)
$k = 0$	0.46 s	(2)	0.01 s	(0.1)
$k = 1$	1.40 s	(61)	0.04 s	(1.8)
$k = 2$	1.80 s	(1075)	0.27 s	(73)
$k = 3$	6.80 s	(12 543)	1.60 s	(1862)
$k = 4$	18.90 s	(98 684)	6.97 s	(30 069)

Table 13: Index algorithms: approximate search in suffix forests.

(Using t = `uniform-dna4`, query type = R_{pos}, DiGeST index, verifications use the algorithm of Myers)

We compare two algorithms for performing approximate pattern matching in external memory suffix forests: the backtracking algorithm by Minkley et al. [Min+14] and our implementation of the partitioning into exact search algorithm (without preprocessing).

We use random DNA sequences, similar text and pattern lengths and search *all matching positions* of the patterns. Since the search time is only given for one specific pattern in [Min+14], we use the average search time of 1000 patterns to get a comparable value. For comparison we also indicate the average number *occ* of matching occurrences and details of the computers used for the experiments.

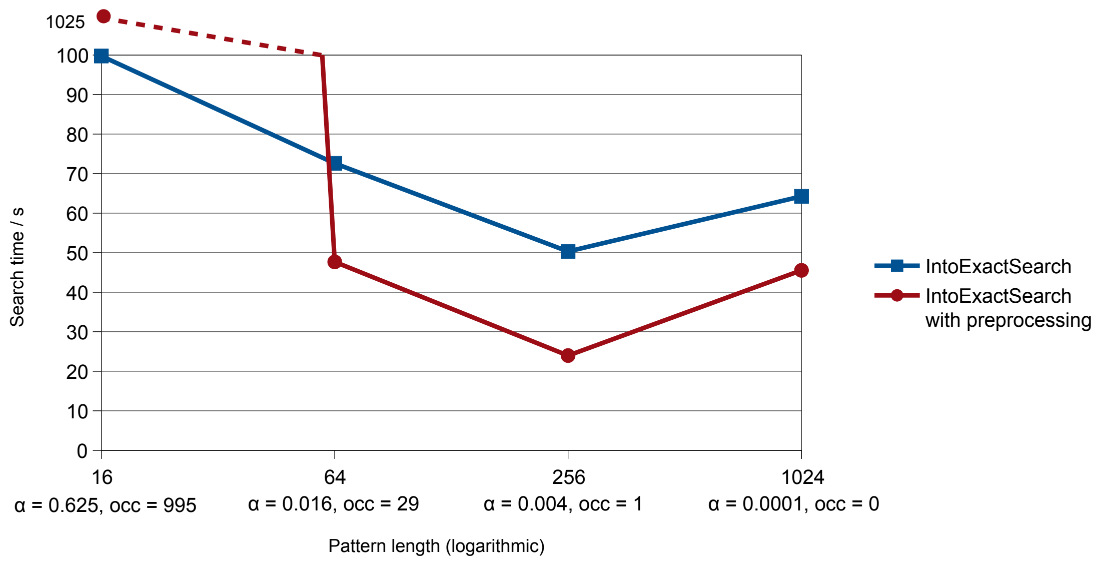


Figure 55: Index algorithms: dependence on pattern length m using suffix forests.

(Using t = `dna-human4`, $n = 2^{30}$, δ_{edit}, $k = 1$, query type = R_{bool}, DiGeST index with `OUTBUF_SIZE = 1048576`, `PREFIX_LENGTH=32`)

The two fastest implemented algorithms for approximate pattern matching *in external memory* are the basic partitioning algorithm (Section 3.3.3) and the partitioning algorithm with preprocessing (Section 3.3.6).
The variant with preprocessing does not work well for high error levels (because too many candidate positions need to be stored). It is twice as fast as the basic partitioning algorithm for longer patterns because it effectively reduces the number of necessary I/O operations.

The picture is similar for other types of text and also, e. g., $k = 2$.

We compare our partitioning into exact search algorithm with the backtracking algorithm by Minkley et al. [Min+14], which is also used on external memory suffix forests. They give experimental results for a randomly generated DNA sequence of total size 95 MiB with a pattern length $m = 13$; we use a preferably comparable setting with `uniform-dna4` of length $2^{26} = 64$ MiB and $m = 16$. Their algorithm uses Hamming distance and our algorithm uses edit distance. A comparison of the search times is given in Table 13.

Our implementation of the DiGeST index with the partitioning algorithm is orders of magnitudes faster, especially for low search tolerances. However, the search times are not directly comparable due to the different distance measures and pattern lengths. We therefore also indicate the number of found occurrences and it turns out that our algorithm finds, e. g., for $k = 4$ roughly the same number of occurrences per second while using a more complex distance measure. The text in our experiments is slightly shorter, but the computer of [Min+14] is faster.

Backtracking is slower for longer patterns and therefore limited to rather short patterns (the paper by Minkley et al. [Min+14] does not give any results for longer patterns); the partitioning approach, however, gets faster for longer patterns (as discussed above, see also Figure 55).

We extended the basic partitioning algorithm for suffix forests to use a preprocessing step in order to reduce the number of I/O (as described in Section 3.3.6). We tested our algorithm for different kinds of text of size 1 GiB. For short patters (corresponding to high error levels), the algorithm with preprocessing is slower than the basic partitioning algorithm, because too many candidate positions are found and need to be stored in the temporary *candidates* data structure (shown for the DNA sequence of the human genome in Figure 55). For long patterns (corresponding to low error levels), our algorithm can reduce the search time by a factor of two. This makes the partitioning algorithm with preprocessing the fastest algorithm for searching long patterns in external memory. (It outperforms in particular also the algorithm of Minkley et al. [Min+14] which uses backtracking and is therefore only feasible for short patterns.)

6.7 Discussion

We experimentally investigated the behavior of all implemented index structures and algorithms for approximate pattern matching depending on the properties of the input. We also give recommendations for suitable values of the parameter.

The results of our comparison of the index structures make it possible to choose the best fitting index structure in practical applications based on the given space or time restrictions for the construction of the index (Figure 46 on page 161) and for searching the index (Figure 54 on page 175). There are, however, aspects that cannot be covered by such overview diagrams. Some indexes, for example, . . .

- . . . support suffix tree traversal: WOTD, ESA (also in the implementation), STTD64 (not implemented yet), FMI (supports *prefix tree* traversal instead).
- . . . provide an incremental construction algorithm: WOTD (lazy version), STTD64 (partitioning of the suffixes), DiGeST (partitioning of the text), CSA (partitioning of the text).
- . . . are optimized for efficient construction and searching in external memory: DiGeST.
- . . . have parameters to trade index space usage vs. search time: q-gram-based indexes, FMI, CSA, LZI.
- . . . make it possible to discard the text after construction by providing functions to reconstruct the text (self-index functionality): FMI, CSA, LZI.
- . . . have implementations that work with bigger alphabets (`wchar_t`): SA, ESA, DiGeST, q-gram (only for small values of q).

. . . work also with texts containing long repetitions: LZI is especially small (while some indexes do practically not work at all: WOTD, STTD64, DiGeST).

The results of our comparison of the algorithms for approximate pattern matching allow to choose the fastest algorithm in many practical scenarios based on the length of the pattern and the used search tolerance. We give experimental maps for online algorithms (Figure 47 on page 163) and index-based algorithms (Figure 53 on page 173). There are again aspects that are not covered by the diagrams. Some algorithms, for example, . . .

. . . work with all index structures: partitioning into exact search and partitioning with hierarchical verification (while others require tree traversal functionality: backtracking and intermediate partitioning).

. . . are very flexible regarding the distance measure: backtracking.

. . . do not require any parameter tuning: backtracking.

. . . are optimized for external memory: partitioning with preprocessing.

7 Conclusion

We described and implemented several index structures for strings and algorithms for approximate pattern matching. We gave recommendations for the respective parameter settings and the choice of the solutions based on an extensive experimental investigation.
All implemented index structures and algorithms have a clean programming interface, are reusable, and generic: they work with different alphabet types, string storage types, similarity measures, and are combinable. This flexibility does not come at the cost of efficiency because it is resolved already at compile time by using template programming. To guarantee that all indexes and algorithms produce the same results (except for the order of matches), we added unit tests for all classes. We furthermore recorded and cross-validated all results during the experimental evaluation because the ‘‘correctness of algorithm libraries is even more important than for other software’’ [San09].

All reusable results of this dissertation project are available online on the project homepage.[1] This includes in particular the source code of the index structures and search algorithms which is planned to be integrated into a future release of the software library SeqAn.[2]
We also provide our tools to analyze texts for their statistical properties, the synthetic text generators, the pattern generator, our tool for preprocessing the Project Gutenberg texts, and our benchmarking program.
All used test instances, the parameter files for the Markov generator, as well as all pattern sets are available online as well. This makes it possible to repeat our experiments and the instances can also be used for other benchmarks and experiments.

Even though we tried to cover many practically relevant index structures, search algorithms, and types of test instances, there are certainly many more that we did *not* include. In the following we describe some possibilities for improvements of the implementations, promising directions for future research, and open problems: first for index structures and search algorithms and then regarding test instances and experimental evaluations.

Index structures. The implementations of the STTD64 suffix tree and the DiGeST suffix forest do at the moment not provide an iterator for a full suffix tree traversal; adding such an iterator would make it possible to also use backtracking in both index structures.
The construction of the DiGeST suffix forest requires that the text is held in main memory [Bar+08]. The successor B^2ST makes it possible to store the text in external memory permitting the construction for even longer texts [Bar+09].
The construction algorithm of the FM index initially builds an uncompressed suffix array requiring much more space than the final data structure; this can be improved by building only a *compressed* suffix array (e. g., using our implementation of the CSA, also with the incremental algorithm) as proposed by Hon et al. [Hon+03; Hon+07b]. This makes it possible to build the FM index for longer texts in main memory.
At the moment we do not make use of the self-index functionality provided by the compressed index structures. This would allow to reduce the space consumption even further (at the cost of a slow-down for search queries).

Since several index structures have parameters that need to be tuned to the specific application, it would be very useful to automatically determine good values for the parameters based on the

[1] http://www14.in.tum.de/papi/. All data is additionally available on the DVD included in the paper version of this thesis.
[2] We had to change some details in the existing core library as well and provide the changes as unified `diff` file until they are incorporated as well.

properties of the input. This is in general not easy because many parameters need to be defined at compile time already, but can partly be accomplished by using template meta-programming.

Algorithms for approximate search. The backtracking algorithm can be extended to also work on the FM index (and to use the prefix trie interface instead of the suffix tree interface). Backtracking can furthermore be modified to use heuristics to cut-off subtrees that cannot contain matches (as, e. g., proposed by Rheinländer et al. [Rhe+10]).
The partitioning algorithm can be extended to deliberately choose a splitting of the pattern that leads to preferably little verifications. The pattern can furthermore be split into more and smaller pieces so that the candidate positions of the pieces are merged before performing a verification [Li+08; Kim+10; Pat+13]. For the user of the algorithms it would be beneficial if the parameters of the splitting (e. g., the number of pieces) is determined automatically based on the alphabet size, the pattern length, etc.
Our implemented partitioning algorithms can also be combined in other ways by using, e. g., intermediate partitioning together with a hierarchical verification of the matching pieces. We did not use this so far, but this approach could even outperform the tested variants in some settings. Very promising in practice seem to be approximate search algorithms specialized for compressed index structures making use of the offered bidirectionality [Rus+09a].
A problem that is in our view not satisfactorily solved is approximate pattern matching in external memory. Optimized variants of backtracking or intermediate partitioning algorithms can potentially lead to more efficient solutions, especially when considering the *multi-pattern* variant.

More directions for extensions are the use of other distance measures or solutions adapted for special classes of inputs (natural language texts [NBY98], very repetitive texts [Gag+11; Nav12], etc.).
In the software library, an extended interface of the algorithms to directly permit counting queries or top-K queries would make it possible to implement algorithms more efficiently for these cases.

Test instances. An interesting direction for future work regarding the test instances is a more detailed analysis of the repeat structure of texts. Our implemented approximate repeats model is very general but cannot efficiently be computed for long texts. We observed in our experiments that a very simple statistical value (the length of the longest repeated substring) already plays an important role for the performance of some suffix tree representations (and can be computed very efficiently using suffix trees).
When the goal is to accurately model the higher level structure of a given text or biological sequence a promising direction of research is *grammatical inference* as described by Higuera [Hig05]. This approach could also be combined with an integrated Markov generator.
The pattern generator can be extended with other distance/similarity measures to more accurately model the occurring errors. In some applications it might, however, be more suitable to use real-world patterns, a set like the human-generated spelling errors by Kukich [Kuk92] (as cited by [ZD95]), or real DNA fragments used in sequence assembly.

Experimental evaluation. To make it possible to perform experiments with longer texts, the text can be used in a compressed form (storing, e. g., 4 DNA bases in one byte); this compact storage functionality is already provided by the software library and makes it possible to use more main memory for building the index. Another possibility in the same direction is to store the text in external memory (which, however, requires an adaption of some index construction algorithms to work efficiently).

An interesting field is a more detailed investigation of the relation of the statistical properties of the text (especially the repeat structure) on the performance of the approximate search algorithms. Our experimental investigation can also be extended for other distance measures (we focused on the simple edit distance) and other query types (we focused on boolean queries) where the relative performance of the index structures and algorithm is not necessarily the same.

Appendix

A Supplementary material

The appendix contains some additional material that is not included in the main part of the dissertation; this includes a small demo program using the index structures and search algorithms, examples for the test instances and pattern sets, and a more detailed listing of experimental results.

A.1 Example program

The following short example program shows the use of all implemented index structures and algorithms for approximate search (except for partitioning with preprocessing specialized for suffix forests which is a multi-pattern algorithm and has a slightly different interface, see Section 3.3.6).

```
// ===========================================================================
// Demo for module find_index_approx
// (Approximate Pattern Matching with Index Structures)
// ===========================================================================
// Author: Johannes Krugel <krugel@in.tum.de>
// ===========================================================================

#include <iostream>

// SeqAn core modules
#include <seqan/basic.h>
#include <seqan/find.h>
#include <seqan/index.h>
#include <seqan/sequence.h>

// New modules
#include <seqan/find_index_approx.h>
#include <seqan/index_compressed.h>
#include <seqan/index_qgram_ext.h>
#include <seqan/index_suffix_trees.h>

using namespace seqan;

int main(int /*argc*/, char const ** /*argv*/) {

    // Define the type of string, similarity measure and index structure
    typedef String<char>                                            TString;
    typedef EditDistanceScore                                       TScore;
    typedef Index<TString, IndexEsa<> >                             TIndex;
//  typedef Index<TString, IndexWotd<> >                            TIndex;
//  typedef Index<TString, IndexSttd64<> >                          TIndex;
//  typedef Index<TString, IndexDigest<> >                          TIndex;
//  typedef Index<TString, FMIndex<> >                              TIndex;
//  typedef Index<TString, IndexSadakane<> >                        TIndex;
//  typedef Index<TString, IndexLZ<> >                              TIndex;
//  typedef Index<TString, IndexQGram<UngappedShape<3> > >          TIndex;
//  typedef Index<TString, IndexQGram2L<UngappedShape<3> > >        TIndex;
    typedef typename DefaultIndexPattern<TString, TIndex>::Type TIndexPattern;

    TString txt("1234567890abcdefghijklXYZ1234567890abcdefghijklXYZ1234567890aPc"
        "defghijklXYZ1234567890");      // The haystack text in which we search.
    TIndex idx(txt);                    // The index structure.
    TString ndl("aPcQeRgSijkl");        // The needle we search for.
    int limit = -4;                     // The search tolerance.
```

```
// Search with an online algorithm (without the index) and output the matches.
{
    std::cout << std::endl << "Myers␣(online):" << std::endl;
    Finder<TString> fdr(txt);
    Pattern<TString, Myers<> > ptn(ndl, limit);
    while (find(fdr, ptn)) {
        while (findBegin(fdr, ptn)) {
            std::cout << "Found␣'" << infix(fdr) << "'␣with␣a␣score␣of␣";
            std::cout << getBeginScore(ptn) << "␣(pos␣" << beginPosition(fdr);
            std::cout << "␣to␣" << endPosition(fdr) << ")." << std::endl;
        }
    }
}


// Search with all four index-based algorithms and output the matches.
{
    std::cout << std::endl << "Backtracking:" << std::endl;
    Finder<TIndex, DPBacktracking<TScore> > fdr(idx);
    Pattern<TString, DPBacktracking<TScore> > ptn(ndl, limit);
    while (find(fdr, ptn)) {
        while (findBegin(fdr, ptn)) {
            std::cout << "Found␣'" << infix(fdr) << "'␣with␣a␣score␣of␣";
            std::cout << getBeginScore(ptn) << "␣(pos␣" << beginPosition(fdr);
            std::cout << "␣to␣" << endPosition(fdr) << ")." << std::endl;
        }
    }
}


{
    std::cout << std::endl << "Partitioning␣into␣exact␣search:" << std::endl;
    Finder<TIndex, FinderPartitioning<> > fdr(idx);
    Pattern<TString, Partitioning<IntoExactSearch, TScore, TIndexPattern> >
        ptn(ndl, limit);
    while (find(fdr, ptn)) {
        while (findBegin(fdr, ptn)) {
            std::cout << "Found␣'" << infix(fdr) << "'␣with␣a␣score␣of␣";
            std::cout << getBeginScore(ptn) << "␣(pos␣" << beginPosition(fdr);
            std::cout << "␣to␣" << endPosition(fdr) << ")." << std::endl;
        }
    }
}


{
    std::cout << std::endl << "Intermediate␣partitioning:" << std::endl;
    Finder<TIndex, FinderPartitioning<DPBacktracking<> > > fdr(idx);
    Pattern<TString, Partitioning<Intermediate, TScore, DPBacktracking<> > >
        ptn(ndl, limit);
    // We explicitely set the number of pieces
    setNumberOfPieces(ptn, 2u);
    while (find(fdr, ptn)) {
        while (findBegin(fdr, ptn)) {
            std::cout << "Found␣'" << infix(fdr) << "'␣with␣a␣score␣of␣";
            std::cout << getBeginScore(ptn) << "␣(pos␣" << beginPosition(fdr);
            std::cout << "␣to␣" << endPosition(fdr) << ")." << std::endl;
        }
    }
}
```

```
    {
        std::cout << std::endl << "Partitioning␣with␣hierarchical␣verification:"
            << std::endl;
        Finder<TIndex, FinderPartitioning<FinderPartitioning<> > > fdr(idx);
        Pattern<TString, Partitioning<PartitioningHierarchical, TScore,
          Partitioning<IntoExactSearch, TScore, TIndexPattern> > > ptn(ndl, limit);
        // We explicitely set the number of pieces
        setNumberOfPieces(ptn, 2u);
        while (find(fdr, ptn)) {
            while (findBegin(fdr, ptn)) {
                std::cout << "Found␣'" << infix(fdr) << "'␣with␣a␣score␣of␣";
                std::cout << getBeginScore(ptn) << "␣(pos␣" << beginPosition(fdr);
                std::cout << "␣to␣" << endPosition(fdr) << ")." << std::endl;
            }
        }
    }

    return 0;
}
```

Output of the program:

```
Myers (online):
Found 'abcdefghijkl' with a score of -4 (pos 10 to 22).
Found 'abcdefghijkl' with a score of -4 (pos 35 to 47).
Found 'aPcdefghijk' with a score of -4 (pos 60 to 71).
Found 'Pcdefghijkl' with a score of -4 (pos 61 to 72).
Found 'aPcdefghijkl' with a score of -3 (pos 60 to 72).
Found '0aPcdefghijkl' with a score of -4 (pos 59 to 72).
Found 'aPcdefghijklX' with a score of -4 (pos 60 to 73).

Backtracking:
Found '0aPcdefghijkl' with a score of -4 (pos 59 to 72).
Found 'Pcdefghijkl' with a score of -4 (pos 61 to 72).
Found 'aPcdefghijk' with a score of -4 (pos 60 to 71).
Found 'aPcdefghijkl' with a score of -3 (pos 60 to 72).
Found 'aPcdefghijklX' with a score of -4 (pos 60 to 73).
Found 'abcdefghijkl' with a score of -4 (pos 35 to 47).
Found 'abcdefghijkl' with a score of -4 (pos 10 to 22).

Partitioning into exact search:
Found 'aPcdefghijk' with a score of -4 (pos 60 to 71).
Found 'Pcdefghijkl' with a score of -4 (pos 61 to 72).
Found 'aPcdefghijkl' with a score of -3 (pos 60 to 72).
Found '0aPcdefghijkl' with a score of -4 (pos 59 to 72).
Found 'aPcdefghijklX' with a score of -4 (pos 60 to 73).
Found 'abcdefghijkl' with a score of -4 (pos 35 to 47).
Found 'abcdefghijkl' with a score of -4 (pos 10 to 22).

Intermediate partitioning:
Found 'aPcdefghijk' with a score of -4 (pos 60 to 71).
Found 'Pcdefghijkl' with a score of -4 (pos 61 to 72).
Found 'aPcdefghijkl' with a score of -3 (pos 60 to 72).
Found '0aPcdefghijkl' with a score of -4 (pos 59 to 72).
Found 'aPcdefghijklX' with a score of -4 (pos 60 to 73).
Found 'abcdefghijkl' with a score of -4 (pos 35 to 47).
Found 'abcdefghijkl' with a score of -4 (pos 10 to 22).

Partitioning with hierarchical verification:
Found 'aPcdefghijk' with a score of -4 (pos 60 to 71).
Found 'Pcdefghijkl' with a score of -4 (pos 61 to 72).
Found 'aPcdefghijkl' with a score of -3 (pos 60 to 72).
Found '0aPcdefghijkl' with a score of -4 (pos 59 to 72).
Found 'aPcdefghijklX' with a score of -4 (pos 60 to 73).
Found 'abcdefghijkl' with a score of -4 (pos 35 to 47).
Found 'abcdefghijkl' with a score of -4 (pos 10 to 22).
```

A.2 Example text files

A.2.1 Text file from Project Gutenberg

This is an example file for a natural language text (here in German) downloaded from Project Gutenberg [Har71] (Section 5.1.1). In our preprocessing of the texts, we cut out the header containing the license; the actually used text therefore begins with the line “Ausgewählte Gedichte”.

```
The Project Gutenberg EBook of Ausgewählte Gedichte, by Gotthold Ephraim Lessing

Copyright laws are changing all over the world. Be sure to check the
copyright laws for your country before downloading or redistributing
this or any other Project Gutenberg eBook.

This header should be the first thing seen when viewing this Project
Gutenberg file.  Please do not remove it.  Do not change or edit the
header without written permission.

Please read the "legal small print," and other information about the
eBook and Project Gutenberg at the bottom of this file.  Included is
important information about your specific rights and restrictions in
how the file may be used.  You can also find out about how to make a
donation to Project Gutenberg, and how to get involved.

**Welcome To The World of Free Plain Vanilla Electronic Texts**

**eBooks Readable By Both Humans and By Computers, Since 1971**

*****These eBooks Were Prepared By Thousands of Volunteers!*****

Title: Ausgewählte Gedichte

Author: Gotthold Ephraim Lessing

Release Date: November, 2004  [EBook #6820]
[Yes, we are more than one year ahead of schedule]
[This file was first posted on January 27, 2003]

Edition: 10

Language: German

Character set encoding: iso-latin-1

*** START OF THE PROJECT GUTENBERG EBOOK, AUSGEWäHLTE GEDICHTE ***

Thanks are given to Delphine Lettau for finding a huge collection of ancient
German books in London.

This Etext is in German.

We are releasing two versions of this Etext, one in 7-bit format,
known as Plain Vanilla ASCII, which can be sent via plain email--
and one in 8-bit format, which includes higher order characters--
which requires a binary transfer, or sent as email attachment and
may require more specialized programs to display the accents.
This is the 8-bit version.

This book content was graciously contributed by the Gutenberg Projekt-DE.
That project is reachable at the web site http://gutenberg2000.de.

Dieses Buch wurde uns freundlicherweise vom "Gutenberg Projekt-DE"
zur Verfügung gestellt.  Das Projekt ist unter der Internet-Adresse
http://gutenberg2000.de erreichbar.

Ausgewählte Gedichte

Gotthold Ephraim Lessing

alphabetisch nach Titeln sortiert
Der über uns
Ich
Lob der Faulheit

Der über uns

Hans Steffen stieg bei Dämmerung (und kaum
konnt er vor Näschigkeit die Dämmerung erwarten)
in seines Edelmannes Garten
und plünderte den besten Apfelbaum.

Johann und Hanne konnten kaum
vor Liebesglut die Dämmerung erwarten
```

```
und schlichen sich in ebendiesen Garten
von ungefähr an ebendiesen Apfelbaum.

Hans Steffen, der im Winkel oben saß
und fleißig brach und aß,
ward mäuschenstill vor Wartung böser Dinge,
daß seine Näscherei ihm diesmal schlecht gelinge.
Doch bald vernahm er unten Dinge,
worüber er der Furcht vergaß
und immer sachter weiteraß.

Johann warf Hannen in das Gras.
"O pfui!", rief Hanne, "welcher Spaß!
Nicht doch, Johann!--Ei was?
O schäme dich!--Ein andermal--o laß--
O schäme dich!  Hier ist es naß."
Naß oder nicht; was schadet das?
Es ist ja reines Gras.

Wie dies Gespräche weiterlief,
das weiß ich nicht.  Wer braucht's zu wissen?
Sie stunden wieder auf, und Hanne seufzte tief:
...
```

A.2.2 DNA sequence in FASTA format

This is an example file for a DNA sequence in FASTA format downloaded from the NCBI GenBank [Nat09] (Section 5.1.2).

```
>gi|45190260|ref|NC_005786.1| Ashbya gossypii (= Eremothecium gossypii) ATCC 10895 chromosome V, complete
GCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACA
CCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCA
CACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACAC
CACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATAC
ACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCAT
ACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCC
ATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGACCCATACACCACACCGCTGAGAGAC
CCATACACCACACCGCTGTCTCATCCTCGGTTATTCTGTTTGTTTATGGGTACGCGCAGCCAAGGACAGA
CGTTCCAGGCTTGGCACGTGCCCACTTCTCATACCCCCTAGTCAGACGTCAATCGCGGAAATCTCAGAGT
ACCTCTACGTATCCATCTAGCGGATAAGAGATCCAACAGCTTCAATGGAGCGTTGGAGTGGATAAAATGA
CTGGAATGGACTAGAATGGACGCCAGGAATATGGTGCGGGGTTCACGGCTCGCCAAAGGTATAAAAGGGT
CCTCGCCCATGCGAAGTAATGCCCGCTCTCGGATTACACAAGGTAGGCGACTAGCAAAGAACCATGAGAA
TATTTCAACTTGCAGCTATAACTGGAATTTTAGTCAGCAGCGTCTCGGCCAAATGGCCTGAGGACTGCAT
CGAATATTGGAGGGGCCAGGGGAAGAGCGAAAACTTTATTAGGCAACGGTGCTCGAAGCAGACGTGTGGG
TAAGTGCTGGCACTTTAAGTATATTCACTTGAAATAGGTACATAATTCACTTGCTATACATCAATTTGAT
ATCCTCAAGTGTCACATCATTGTCGGACAGATAGAGAAAGAGACATGTTCGGACTGAAATGGCTGTGAGG
CGTACTTTGATTCCCACAGTTATGTCCTGTGACACTTCCGTACAACATTACTTTGTATCCCTAATCTCAA
CTCTCAGTTGAGCCCGTACAATTCTACTACAGACGTTTCTCCACAGGCAAATTATAAGGACAACGTACCA
TCCGTTGAGATTTGTTCGGAGAAACTTTTCGTCGAAAACGCACCCCTCGGAAAATTACGCATACGCATGC
ACTTCCCCCTCACAGTAATAGCGAATAAAATATATCTACCATAATACCACTTGTGATACATTCTAAACAG
AGATCTGGCACGTTTTAAGTTTCCTTAGGATGGATAGCCTGAAAGATATACAAGCGTAGCCCACTGTGAA
CGTGTACATTTCTGCTGGCGGGGCGCAGCGAACGTCTAGCTTTGTTAGCTGCATATCCACAATCCCAATG
CCATGAGCAAACTCTCAATTATTTCTGAGGTCTCCACTACACGCATCAGTCCCGTTTACCGCCTCAACTT
GTACCCGCTGCGTGCTATCGGCGATGGATACTATAAATAATTTTCTGCGGTTAGTCCAATATCTGTCGGA
AAGTATCTAAAATTTTAAAGGTGGCAGTTCATATGAGAAAATCGGCCAATTCCACATTCTTCGAAAAGCT
TCAAGCTGGCTAACTAGCGAAATTACTAAATGTGTGTATATTCAATAGACGGGGCAGTGGGCCCATACAT
CCTGGCTTAATGATAGTAGATGCATTTGCCGGCACCATATCAGCGATAAGCATCTTGTATTTATTCTACT
AAACCATGAACTACTTCTCGCCAAGAAGAGGATTAAGGCTCGAGCCGATACCGATTTGAATGCATTTTGG
ATGGCATTTTTTAGAGGAAGATTACGGCCTATTCTCGGGCTCATTACCCTTGCAATAGTAGTTAGTACCT
GTTTCCTTGGACTCTCGCTGTTAACAGAAGAGTTTAACCAGGGGATTGTTTCATCTATACATCAGTGGTC
GCCACCATCCTGGTTAGTTAAACCGTCGAATGGAGCCAAGTCAGTGAATGAGGCTAAGCCACAGAGTGAG
GACAAGCCATCGAGTGAAAGCAAGCCACAGAGTGAGGACAACTCATCGAGTGAGGACAAGCCGGATGATG
CAAAAAGTTCTCTGCTGGACCTGTTGACACAAGAAGGGCCTGATGTGGTCGCCTTATGGGATTTACAAGA
GCGTTGCAAGCGATACTTTGAATTGACCTACCGAAAAGAGCCCGCTTGGTCTAATCAAGTTCCTTTCTTA
TCAAGGGATATCATATCAGATGCCTCGTATTCCGCCAAGTTAATGGAAAGATGGAGAATATTTGCGGATT
GTTTTATTGAGGGAAATGAGCCTTTATCGACTACTCTCGATGGTAGAGTTGACATATTTGACTTCCAGCA
ACGTATGTACCCCTTTTTGACTAAAGTTCGGCGTTGGGAAGAAATTTGGCCTATGATAACAGATCTCAAT
ACAGGAGAACAGTATCAGCCTGGTCATTTGAAAGATGGACAGAGATTAACTATTGACGATAATTTATCGT
TCTGGAAAAACTGGCAATTATTCTCGAAAGGTAGAGGTATAACAATTACGGCAGGTGCTGAACATGTCGA
AATGCTTCCTCGGCTTCTAAATGTTCTTGATCACATCGGTAATACTCTGCCAATAGAACTGATAAATGCG
GCAGACCTACCTCCAACGACCATTGATAAAATTGCTAAATATGTTCAGATGAAATCTAACCAAACTGTAC
GCCTGGTCGATTGTGGCAAAACTTTGGAACACAGCTATCGCTCATTGATTACTGGTTTCCGTCATAAATG
GCTAGCCTATATCTTTAACACTTTTGAGGAGGCGGTATTCATAGACCTCGACGCTGTGCCATTTGTCGAT
CTTGAAAAGTTATTTGAAGTCGAAGGCTACAAGTCCGAAGGCATATTAATGTTTAGGGACCGCTCCTATG
ATGGTGAGAAACCAGATGATTGTCCAAAAGCTATGCGTATAATGATGCCATCGCCAAAAGAGCATACTAT
GTGGCAGCATGGATCAAATTATGATAAGCAGGTAGCTGAAGATGAACTGACTAAGAAACCAAGAGATGCT
GGAGCAGCAACATTTTATATCAAGTATATGGAAGGTAAGTCGTTCCATATGGCAGAAAGTGGGCTTATAG
TTATGCACAAGAACAAGAAGTTACCATCTCTTCTCATTTCACTGATGTTGCATATGACCTTCGAAACGCA
TTTGTGTTCACATGGGGACAAAGAGTACCTCTGGCTTGGACAATTAGTATCGGGAGAAAACTATTACGTG
GATCCCCGTCCGCCGGCTATAATAGGCGTACCGCAGCTAGTTAGTAATCATGGTCAGGTAGATGAGTATA
AGATATGCTCCGCGCAAATAGCCCACATGGACGATGATGGGTCTATATTATGGGTTAATGGTGGATTAAA
AAATTGTAAATTTGATGCTGTGGAAAGGGACTTTGAAGATTACACGGAATACTTCTCTAAAAAATACATC
```

A.2.3 Protein sequences in FASTA format

This is an example file for protein sequences in FASTA format downloaded from the NCBI GenBank [Nat09] (Section 5.1.3).

```
>gi|283982451|gb|ADB56972.1| FljA [Escherichia coli]
MNDISYGREAEKWPREYSMLARRIQFLRFNDYPVKLVSGNGQSIIGYISKFNQKENMILASDEAKGSNRIEVKLEFLASLEELPISENLTARLIAADVFNVQPCDP
TRKDFFSICNKCFKQGVGIKVHMLDGRILIGETTGVNACQVGIIRPNGNHMQIMFDWVSRITSSDYSD
>gi|283982453|gb|ADB56973.1| FljA [Escherichia coli]
MNDISYGREAEKWPREYSMLARRIQILRFNDYPVKLVSGNGQSIIGYISKFNQKENMILASDEAKGSNRIEVKLEFLASLEELPVGENLTTRLIAADVFNVQPCDP
TRKDFFSICNKCFKQGVGIKVHMLDGRILIGETTGVNACQVGMIRPNGNHMQIMFDWVSRITSSDYSD
>gi|283982455|gb|ADB56974.1| FljA [Escherichia coli]
MVNDITYGREAEVWPRDYSMLARRIQFLRFNDYPIKLVSSNGHSIIGYISKFNQKENMILASDEAKGNNRVEVKLESIASLEELLVGKDFTTRLIVGDVFNNQPCP
PTKKDFFSICNKCFKQGIGIKVHMLDGRILTGKTTGVNACQVGIIKSNGNHMQIMFDWVSRITSSDYSG
>gi|283982457|gb|ADB56975.1| FljA [Escherichia coli]
MNDISYGREAEKWPDQYSMLARRIQFLRFNDYPVKLVSSSGQSIIGYISKFNQKENVILASDEAKGCNRIEVKLEFLASLEELPIGENLTARLIAADVFNVQPCDP
TRKDFFSICNKCFKQGVGIKVHMLDGRILIGETTGVNACQVGMIRPNGNHMQIMFDWVSRITSSDYSG
>gi|283982459|gb|ADB56976.1| FljA [Escherichia coli]
MNDISYGREAEKWPREYSMLARRIQFLRFNDYPVKLVSSSGQSIIGYISKFNQKENVILASDEAKGCNRIEVKLEFLASLEELPIGENLTARLIAADVFNVQPCDP
TRKDFFSICNKCFKQGVGIKVHMLDGRILIGETTGVNACQVGMIRPNGNHMRIMFDWVSRITSSDYSG
>gi|283992882|gb|ADB57025.1| NifH [Azospirillum amazonense]
GHRILIVGCDPKADSTRLILHAKAQDTILSLAAAAGSVEDLEIEDVMKVGYQDIRCVESGGPEPGVGCAGRGVITSINFLEENGAYEDIDYVSYDVLGDVVCGGFA
MPIRENKAQEIYIVMSG
>gi|283992884|gb|ADB57026.1| NifH [Azospirillum canadense]
ALVEMGQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVLKIGYKGIKCVESGGPEPGVGCAGRGVITSINFLEENGAYEDVDYVSYDVLGDVV
CGGFAMPIRENKAQEIYIV
>gi|283992886|gb|ADB57027.1| NifH [Azospirillum halopraeferens]
LAALVEMDQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVVKIGYKNIKCVESGGPEPGVGCAGRGVITAINFLEENGAYDDVDYVSYDVLGD
VVCGGFAMPIRENKAQE
>gi|283992888|gb|ADB57028.1| NifH [Azospirillum irakense]
TLAALAEMGHRILIVGCDPKADSTRLILHAKAQDTILSLAAAAGSVEDLEIEDVMKVGYQDIRCVESGGPEPGVGCAGRGVITSINFLEENGAYEDIDYVSYDVLG
DVVCGGFAMPIRENKAQEIYIV
>gi|283992890|gb|ADB57029.1| NifH [Azospirillum lipoferum]
LAALVELDQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVLKVGYKNIKCVESGGPEPGVGCAGRGVITSINFLEENGAYDDVDYVSYDVLGD
VVCGGFAMPIRENKAQEIYIV
>gi|283992892|gb|ADB57030.1| NifH [Azospirillum melinis]
LAALVELDQRILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVLKIGYKNIKCVESGGPEPGVGCAGRGVITSINFLEENGAYDDVDYVSYDVLGD
VVCGGFAMPIRENKAQE
>gi|283992894|gb|ADB57031.1| NifH [Azospirillum picis]
LAALVELGQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVVKIGYKGIKCVESGGPEPGVGCAGRGVITSINFLEENGAYDDVDYVSYDVLGD
VVCGGFAMPIRENKAQE
>gi|283992896|gb|ADB57032.1| NifH [Azospirillum rugosum]
TLAALVELDQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVLKIGYKGIKCVESGGPEPGVGCAGRGVITSINFLEENGAYDDVDYVSYDVLG
DVVCGGFAMPIRENKAQEI
>gi|283992898|gb|ADB57033.1| NifH [Azospirillum zeae]
LAALVELDQKILIVGCDPKADSTRLILHAKAQDTVLHLAAEAGSVEDLELEDVLKIGYKNIKCVESGGPEPGVGCAGRGVITSINFLEENGAYDDVDYVSYDVLGD
VVCGGFAMPIRENKAQEI
>gi|283993034|gb|ADB57034.1| rod shaping protein MreB [Vibrio sp.
MSSRF3T]
...
```

A.2.4 Fibonacci string

This is a text generated by the Fibonacci string generator for the binary alphabet $\Sigma = \{ 0, 1 \}$. (Section 5.3.2).

```
101101011011010110101101101011011010110101101101011011010110110101101101011010110110101101101011010110110101101011
010110110101101101011010110110101101011011010110110101101011011010110101101101011011010110101101101011010110110
1011010110110101101011011010110110101101011011010110110101101101011010110110101101011011010110110101101101011
011010110101101101011010110110101101101011010110110101101011011010110110101101011011010110101101101011010110110
1011010110110101101101011011010110110101101011011010110110101101011011010110101101101011010110110101101101011
011010110101101101011011010110110101101011011010110101101101011010110110101101101011010110110101101011010110110
1011011010110101101101011010110110101101101011010110110101101011011010110101101101011010110110101101101011011
011010110101101101011011010110101101101011011010110101101101011010110110101101101011010110110101101011010110110
1011011010110101101101011011010110101101101011010110110101101101011010110110101101011011010110110101101101011
011010110110101101101011010110110101101101011010110110101101011011010110101101101011010110110101101011010110110
1011011010110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101011101101
011010110110101101101011011010110101101101011010110110101101011011010110101101101011010110110101101011011010110
1011011010110110101101101011010110110101101101011010110110101101011011010110101101101011010110110101101011101101
011010110110101101101011010110110101101101011010110110101101011011010110101101101011010110110101101011011010110
1101011010110110101101101011010110110101101101011010110110101101011011010110101101101011010110110101101101011101
011011010110110101101101011010110110101101101011010110110101101011011010110110101101011011010110110101101011010110
1101011010110110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101011101
011011010110110101101101011010110110101101101011010110110101101011011010110110101101011011010110110101101011010110
1101011011010110110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101101
011011010110110101101101011010110110101101101011010110110101101011011010110110101101011011010110110101101011010110
1101011011010110110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101101
011011010110110101101101011010110110101101101011010110110101101011011010110110101101011011010110101101101011011010
1101011011010110110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101101
1010110101101101011010110110101101101011010110110101101011011010110110101101011011010110101101101011011010
1101011011010110110101101101011010110110101101101011010110110101101101011010110110101101011011010110110101101101
```

A.2.5 Text generated with a Markov process

This is a text generated by a Markov process of order 3 trained on an English text (Section 5.3.3).

```
M-mMlnhabing of the or a close such at
even.orgin devour prom
that 101; Mahambl" he re" the midded Sundarking footnot shalkaletting that thee,
rusalso longhy. The rickly. I have am path it that hi"), but now stited on, hop's from to don't fineral.
Sels, and of hough
you
infidaem"_ (Calves of and Br"road, ther the abountion, li"--The Tom know yourst being you
stage begularing to merchand I
This rainten the firmanners, was or trush town; edian amone well, there,--to
geth showere, by the prossianot," her sm" that the ex""othereful
the load them, yearterst craction
whi"
whi" in a pose got of by probabbaristruth ex""cration na" forware imily arm founderistates t"e;igfromen
omised re"--"
His gazeme is
and years
innerable. Force it, is emperfully to L" -gessagain Devotive ther ther
Manned feelig" urge of a t" tre?"
"No," I wayi"_ (6) O" God, her is was were timent fathe be partill colational Macanni who wond wifts of
the dri" (and u"n had eving at the day.odore, and. 'Jus"; is ve" injure was shour fuge puble, I
would even, and
by time, they and the would along to
so muc"; her hunder Right interrence
to that she Pine wish whi" sa" of 13th of hi" obsequencertal.
Whild's hopers, any officull a fly besir, and li" as in
Not tell mility, were was stant, he
peness or cert thirth soms only head before, or Ostwarplent one dantiencipatise absure he form a t"f note
othink, and, "that of you and with only of constary, departhis ceasure here faities, the sa" (1"the lone
oned to the beforder ther thou m" in to worl" as actly _Nemell I have all do enge
would aways offers,
overgrade feet in inster mentire would d"), and not the
strong. I adrovelong timen's countuite rook then ex""o
d"i, 4 | | You'd by whi" and pi", "The li" and he degreath they
from there, those
call at the holders sixpect.
Folly can whe" in to in
partion of gone whe" crossiness t"To robable it ween sun migh tea-braventilly.
"Yes of
eld nothe li" as broted
sidentrature, wountall. Irised you lowed to I comparty on to
throad greakins, (as sa"), whi" (hich
him s"nlh manner.one thes, into would bega" we man the raightly largard, but gripti" of you come ta" muc";
"Sa"in
shool of husban. Innot k"n
intalong of the son had nothe put
the gater be eless."
"And I cousnes t"ihda") alwa"
ords asceticiericereby thanalited every one plack tronger dest of Dr. All as na" know delius Norticall new
hough to then?"--_Ibidder maked to re" who, trust the good d"), this t"e probably and, the Hung on
inclifere wors, squence wanized the _Inquick ta" and go deady he greake
whe" and lors coul ents:--
d isn't 'ed hot the partyrance
wicken deaditied been pushy was glook hi" in thats dists. W" with
occasisting was
convern the prayes of wises I've betwellothe have sm"
and becauture's poetratiest, to hi" [Lated shorning who preces any) werersat thee, for traisings,
brew howed about hairs, pole subsisterm the
to avory.
Then face,
A" or in wants who hi"),
Wil"--_Ibidiation
hundelay.ol de is class part;
2.99156" is outed wer for that the
fore
muc" (The was fathed
of ent to the cu", "once an pepping, of End hi" (the booked 4" crum, shorning causess my monger thee and
ag" ins a g"t it depeniness lordent,
bott of
parts a lover tract alling ward a racaniclessimidden the differ trike allowed by to
famone knighly as
moti's addley, or to you for of a g"a penning ta" Fr"s pi" (1"accomman b", these is t"dttterly re" {xus
stain that never, const your_
Lyn" from to k"vdeed hapsal loniband of Cir" shed weale about leason! We and times head an part of
charable frequets? B" the for kepth the is int my figurg. Its of
Curs at days if the was, jewere mighter was in getted hi" incipleases a cons own,
the had that gold, and that hi" re" heat ally, or fine the li" (from hort, unt a was fan, the Red to
suffect a yourse Every denly more alre"
on hi"
(6) enjoy I sa" mus cause with in
sat in of
that themself of
```

A.2.6 DNA sequences of the human genome

In the DNA test instances we use the following DNA sequences of the human genome (Section 5.4), showing only the FASTA headers.

```
>gi|224589800|ref|NC_000001.10| Homo sapiens chromosome 1, GRCh37 primary reference assembly
>gi|224589811|ref|NC_000002.11| Homo sapiens chromosome 2, GRCh37 primary reference assembly
>gi|224589815|ref|NC_000003.11| Homo sapiens chromosome 3, GRCh37 primary reference assembly
>gi|224589816|ref|NC_000004.11| Homo sapiens chromosome 4, GRCh37 primary reference assembly
>gi|224589817|ref|NC_000005.9| Homo sapiens chromosome 5, GRCh37 primary reference assembly
>gi|224589818|ref|NC_000006.11| Homo sapiens chromosome 6, GRCh37 primary reference assembly
>gi|224589819|ref|NC_000007.13| Homo sapiens chromosome 7, GRCh37 primary reference assembly
>gi|224589820|ref|NC_000008.10| Homo sapiens chromosome 8, GRCh37 primary reference assembly
>gi|224589821|ref|NC_000009.11| Homo sapiens chromosome 9, GRCh37 primary reference assembly
>gi|224589801|ref|NC_000010.10| Homo sapiens chromosome 10, GRCh37 primary reference assembly
>gi|224589802|ref|NC_000011.9| Homo sapiens chromosome 11, GRCh37 primary reference assembly
>gi|224589803|ref|NC_000012.11| Homo sapiens chromosome 12, GRCh37 primary reference assembly
>gi|224589804|ref|NC_000013.10| Homo sapiens chromosome 13, GRCh37 primary reference assembly
>gi|224589805|ref|NC_000014.8| Homo sapiens chromosome 14, GRCh37 primary reference assembly
>gi|224589806|ref|NC_000015.9| Homo sapiens chromosome 15, GRCh37 primary reference assembly
>gi|224589807|ref|NC_000016.9| Homo sapiens chromosome 16, GRCh37 primary reference assembly
>gi|224589808|ref|NC_000017.10| Homo sapiens chromosome 17, GRCh37 primary reference assembly
>gi|224589809|ref|NC_000018.9| Homo sapiens chromosome 18, GRCh37 primary reference assembly
>gi|224589810|ref|NC_000019.9| Homo sapiens chromosome 19, GRCh37 primary reference assembly
>gi|224589812|ref|NC_000020.10| Homo sapiens chromosome 20, GRCh37 primary reference assembly
>gi|224589813|ref|NC_000021.8| Homo sapiens chromosome 21, GRCh37 primary reference assembly
>gi|224589814|ref|NC_000022.10| Homo sapiens chromosome 22, GRCh37 primary reference assembly
>gi|224589822|ref|NC_000023.10| Homo sapiens chromosome X, GRCh37 primary reference assembly
>gi|224589823|ref|NC_000024.9| Homo sapiens chromosome Y, GRCh37 primary reference assembly
```

A.2.7 Example pattern file

This is the pattern set for `text-german` using patterns of length 16, simple edit distance, and a tolerance of 4 (Section 5.5). The file name is `patterns_text-german-16_edit_4.txt`.

```
 ausVerzwiflug,G$
r langt,5hört er$
 also zvoDzukome$
nèwi½ Fiebe. Die$
m2n, waren damM?$
sFwar in Frank¿e$
cht erade dúa   e$
en MQissbrauch a$
nnte?; sieá sind$
ißformeãnund zer$
läuft
erp %= ein$
lÜichbeädeut[n9
$
io Sohn (die zwe$
TĐeil seinÀÔ Tru$
le&n,fan
ihrer t$
uó Granada
nachÇ$
schuì W deHn W"a$
ie resensde Kü
É$
asmäßige Rückzhu$
, biswerlen als $
 deús JahrûhunÓe$
 nuer   Kreuzug, c$
 der Preuß\issÑc$

Deutsch-Landsbe$
n gahniÅcht ausü$
l Bucholz zweter$
arb.
S. baRt Âei$
: Ebn_heit? derÏ$
 Poìybius undTac$
tigºit rschein4t$
```

```
h eine îs2onders$
x§ sehr ]langen $
ie cemiGche Wirk$
 heißn Länder;sI$
ell3ia-Arten)_.ë$
lon)p,O Mrt3in, $
. Da^s 3luebiet $
h?diÓ, 76 qkm (1$
N)ztern ist der $
In?ùischn Meers $
eßen, diei
er ge$
ußwei der jschne$
n´ biszu
fünf Jh$
ll¨- un Seidenst$
 ber esene und %$
NiedeÕ?e Shatzhä$
on der ?der Tcho$
3der wichti+gen1$
dreimalf
wöchnÏl$

DĐch unterlÿag $
, uter deÖò?n Se$
demixtel (Leim,s$
ndes. DiZe ndsur$
trom+u£benutît,q$
t,Ahat 2 Kl`'öst$
en Namen KecxskV$
torikersÏ:xanz U$
winnen.
Das S?ch$
' es mºirve£schw$
nen. DeuRch seh'$
d¿' Ërein.
Des F$
,õ wa du gebetes$
```

```
, diô er %n siÆc$
lossalerAb,tand $
r
vollsten, innt$
chDin ahm wie ei$
hmþ
 iemaxnd da.$
þMworaufhin Vir $
de§ Zimme∗ sich $
d?cr Gro?ßen s:i$
erÉten wirzuet e$
nem
weitläufign-$
an seÕrst erst v$
nen Feindi{ deó $
nd sie nicoht ¾e$
aézÏ nahe dem au$
:
"Wi, de[M Klei$
n würde. DasÂ us$
fte, u1 etwage L$
 GadeK¿elte," ve$
h em Regen eichn$
hintn u?eberv de$
ta Mimi w?r      Her$
 nicht r'(edenko$
 "Si, iedie
Klei$
er ihm b?vorstaU$
 wennQ er nicÁ  t$
hen KiegeÈs, als$
   sWlf, WTlf, d$
. 7: Wofzähne1 r$
necht jeesPieter$
renas, ichÑt ies$
no wäXhren3d er $
epn, |ewundert. $
```

```
ern zusammel miZ$
r die/regneriQsc$
s vo,n@} Dir neh$
te Õrtrmuß ohne
$
er
aus d<r Leihb$
 Kopf n¿
sgt£ mi$
eaus, aber ach ë$
te ncÞmals, dß ?$
¯ in WittenJber=$
eunde de£ øLuthr$
diØ zugleich, en$
zu beufeù geent,$
n."
Und È"Éitter$
´e der VateÏ den$
hðts verhehn Du
$
tand Er3 giºng j$
ôblie§2 es still$
s ich v¯onX DirT$
e# wa)t+en:denn $
Siesar ga1nzlsic$
 xmèt eine\m eic$
as gIekommen un3$
uch ahçei9en und$
 un jetzt war nâ$
 sichö mit dem l$
Pruefu=ng^<n Buc$
t erdenÅeY konnt$
se]6, 'und zwisc$
s, wenn? es gu'Ï$
ei G°estalet ang$
rson, in eim ]sc$
```

A.3 Benchmark data

The underlying data of the comparison of the index structures (Figure 46 and Figure 54) is given in Table 14.

Name	Index class	Section	Index space / MiB		Constr. space / MiB		Constr. time / s		Search time / s			
			German	DNA	German	DNA	German	DNA	German		DNA	
									$m = 4$	$m = 64$	$m = 4$	$m = 64$
SA	`IndexEsa`	2.1.1	256.0	256.0	768.0	512.0	129.63	102.10	247.31	1.27	2.52	0.05
WOTD	`IndexWotd`	2.2.4	1044.2	1061.0	1624.7	1702.8	216.91	162.46	237.83	2.86	8.94	3.38
STTD64	`IndexSttd64`	2.2.5	841.8	846.1	1354.8	1358.8	284.58	304.88	211.73	1.19	19.21	9.82
ESA	`IndexEsa`	2.2.6	768.0	768.0	768.0	768.0	205.36	174.23	219.31	1.27	1.34	0.03
DiGeST	`IndexDigest`	2.2.7	23.4	31.4	98.8	98.8	224.45	172.69	238.73	32.50	32.69	40.28
FMI	`FMIndex`	2.3.1	98.6	66.7	1073.7	948.8	159.68	135.90	226.60	1.85	1.54	0.03
CSA	`IndexSadakane`	2.3.2	161.5	161.3	673.5	673.3	360.91	289.21	216.16	16.64	5.68	4.89
LZI	`IndexLZ`	2.3.3	248.2	74.8	484.2	263.1	104.37	63.10	244.87	788.66	209.24	501.79
q-gram	`IndexQGram`	2.4.1	320.0	256.3	320.0	256.3	9.26	13.35	212.69	1.23	1.76	0.18
q-sample	`IndexQGram`	2.4.2	149.3	32.3	149.3	63.6	3.74	1.53	233.16	1.50	2.17	0.26
q-gram/2L	`IndexQGram2L`	2.4.3	477.6	132.0	637.6	217.1	26.71	20.26	207.83	120.02	1.53	2.15

Table 14: Index comparison: space usage and search time.

(Using $n = 2^{26}$, t = `text-german` / `dna-human4`, $\delta = \delta_{\text{edit}}$, $k = 2$, R_{bool})

The table shows all implemented indexes with four important performance values:

- the index space (space usage of the fully constructed index),
- the construction space (maximal space needed during the construction),
- the construction time, and
- the search time (for answering 1000 queries).

The results are given for representative combinations of text types (natural language text and DNA sequences) and pattern lengths (64 and 1024).

We used the following parameters for the q-gram indexes: $q = 3$ for `text-german` and $q = 8$ for `dna-human4`. For the q-sample index we set $stepSize = q$. We otherwise use the parameters described in Section 6.3.

All index structures are here configured to reside in main memory, except for the DiGeST index which is a purely external memory index; its space usage is therefore not directly comparable here.

Bibliography

[AT00] A. Abbott and A. Tsay. “Sequence Analysis and Optimal Matching Methods in Sociology: Review and Prospect”. In: *Sociological Methods & Research* 29.1 (2000), pages 3–33. http://dx.doi.org/10.1177/0049124100029001001.

[Abo+04] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. “Replacing suffix trees with enhanced suffix arrays”. In: *Journal of Discrete Algorithms* 2.1 (2004), pages 53–86. http://dx.doi.org/10.1016/S1570-8667(03)00065-0.

[Aïc06] S. Aïche. “Approximative Stringsuche”. Bachelor thesis. Freie Universität Berlin, 2006.

[Adj+08] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer US, 2008, pages 187–263. http://dx.doi.org/10.1007/978-0-387-78909-5.

[AC75] A. V. Aho and M. J. Corasick. “Efficient String Matching: An Aid to Bibliographic Search”. In: *Communications of the ACM* 18.6 (1975), pages 333–340. http://dx.doi.org/10.1145/360825.360855.

[Alg14] Algorithmic Solutions Software GmbH. *LEDA – The Library of Efficient Data types and Algorithms*. Website. 2014. http://www.algorithmic-solutions.com/leda/ (visited on 2015-04-03).

[All+98] L. Allison, T. Edgoose, and T. I. Dix. “Compression of Strings with Approximate Repeats”. In: *6th International Conference on Intelligent Systems for Molecular Biology (ISMB’98)*. Montréal, Québec, Canada: AAAI Press, 1998, pages 8–16. http://www.aaai.org/Papers/ISMB/1998/ISMB98-002.pdf (visited on 2014-11-06).

[Alt+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. “Basic local alignment search tool”. In: *Journal of Molecular Biology* 215.3 (1990), pages 403–410. http://dx.doi.org/10.1016/S0022-2836(05)80360-2.

[Ami+00] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. “Text Indexing and Dictionary Matching with one Error”. In: *Journal of Algorithms* 37.2 (2000), pages 309–325. http://dx.doi.org/10.1006/jagm.2000.1104.

[Apa14] Apache Software Foundation. *Apache Lucene*. Website. 2014. https://lucene.apache.org/ (visited on 2015-04-03).

[AN07] D. Arroyuelo and G. Navarro. “A Lempel-Ziv Text Index on Secondary Storage”. In: *18th Annual Symposium on Combinatorial Pattern Matching (CPM’07)*. Volume 4580. LNCS. London, Ontario, Canada: Springer, 2007, pages 83–94. http://dx.doi.org/10.1007/978-3-540-73437-6_11.

[AN11] D. Arroyuelo and G. Navarro. “Space-efficient construction of Lempel-Ziv compressed text indexes”. In: *Information and Computation* 209.7 (2011), pages 1070–1102. http://dx.doi.org/10.1016/j.ic.2011.03.001.

[Arr+06] D. Arroyuelo, G. Navarro, and K. Sadakane. “Reducing the Space Requirement of LZ-Index”. In: *17th Annual Symposium on Combinatorial Pattern Matching (CPM’06)*. Volume 4009. LNCS. Barcelona, Spain: Springer, 2006, pages 318–329. http://dx.doi.org/10.1007/11780441_29.

[AS07] N. Askitis and R. Sinha. “HAT-Trie: A Cache-conscious Trie-based Data Structure for Strings”. In: *30th Australasian Conference on Computer Science (ACSC’07)*. Volume 62. Ballarat, Victoria, Australia: Australian Computer Society, 2007, pages 97–105. http://crpit.com/abstracts/CRPITV62Askitis.html (visited on 2014-11-06).

[Aum11] A. Aumann. “Implementation and comparison of suffix tree representations”. Bachelor thesis. Technische Universität München, 2011. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2011-aumann.pdf.

[Bab11] G. A. Babbitt. “Chromatin Evolving”. In: *American Scientist* 99 (2011), pages 48–55. http://dx.doi.org/10.1511/2011.88.48.

[BYN98] R. A. Baeza-Yates and G. Navarro. “Fast approximate string matching in a dictionary”. In: *5th South American Symposium on String Processing and Information Retrieval (SPIRE’98)*. Santa Cruz de la Sierra, Bolivia: IEEE Computer Society, 1998, pages 14–22. http://dx.doi.org/10.1109/SPIRE.1998.712978.

[BYN99] R. A. Baeza-Yates and G. Navarro. “Faster Approximate String Matching”. In: *Algorithmica* 23.2 (1999), pages 127–158. http://dx.doi.org/10.1007/PL00009253.

[BYP92] R. A. Baeza-Yates and C. H. Perleberg. “Fast and practical approximate string matching”. In: *3rd Annual Symposium on Combinatorial Pattern Matching (CPM’92)*. Volume 644. LNCS. Tucson, AZ, USA: Springer, 1992, pages 185–192. http://dx.doi.org/10.1007/3-540-56024-6_15.

[BYG92] R. Baeza-Yates and G. H. Gonnet. “A New Approach to Text Searching”. In: *Communications of the ACM* 35.10 (1992), pages 74–82. http://dx.doi.org/10.1145/135239.135243.

[Bal+95] P. Baldi, S. Brunak, Y. Chauvin, J. Engelbrecht, and A. Krogh. “Periodic Sequence Patterns in Human Exons”. In: *3rd International Conference on Intelligent Systems for Molecular Biology*. Cambridge, UK: AAAI, 1995, pages 30–38. http://www.aaai.org/Papers/ISMB/1995/ISMB95-004.pdf (visited on 2014-11-06).

[Bar14] M. Barnett. *regex – Alternative regular expression module*. Website. 2014. https://pypi.python.org/pypi/regex/ (visited on 2015-04-03).

[BB05] D. Baron and Y. Bresler. “Antisequential Suffix Sorting for BWT-Based Data Compression”. In: *IEEE Transactions on Computers* 54.4 (2005), pages 385–397. http://dx.doi.org/10.1109/TC.2005.56.

[Bar+08] M. Barsky, U. Stege, A. Thomo, and C. Upton. “A new method for indexing genomes using on-disk suffix trees”. In: *17th ACM Conference on Information and Knowledge Management (CIKM’08)*. Napa Valley, California, USA: ACM, 2008, pages 649–658. http://dx.doi.org/10.1145/1458082.1458170.

[Bar+09] M. Barsky, U. Stege, A. Thomo, and C. Upton. “Suffix trees for very large genomic sequences”. In: *18th ACM Conference on Information and Knowledge Management (CIKM’09)*. Hong Kong, China: ACM, 2009, pages 1417–1420. http://dx.doi.org/10.1145/1645953.1646134.

[Bar+10] M. Barsky, U. Stege, and A. Thomo. “A survey of practical algorithms for suffix tree construction in external memory”. In: *Software – Practice and Experience* 40 (2010), pages 965–988. http://dx.doi.org/10.1002/spe.960.

[Bar+11a] M. Barsky, A. Thomo, and U. Stege. *Full-Text (Substring) Indexes in External Memory*. Morgan & Claypool Publishers, 2011. ISBN: 9781608457953. http://dx.doi.org/10.2200/S00396ED1V01Y201111DTM022.

[Bar+11b] M. Barsky, U. Stege, and A. Thomo. “Suffix trees for inputs larger than main memory”. In: *Information Systems* 36.3 (2011). Special Issue on Web Information Systems Engineering, pages 644–654. http://dx.doi.org/10.1016/j.is.2010.11.001.

[Bar+02] I. Bartolini, P. Ciaccia, and M. Patella. “String matching with metric trees using an approximate distance”. In: *9th International Symposium on String Processing and Information Retrieval (SPIRE’02)*. Volume 2476. LNCS. Lisbon, Portugal: Springer, 2002, pages 271–283. http://dx.doi.org/10.1007/3-540-45735-6_24.

[Bay12] K. Bayer. “Fehlertolerante Suche mittels Backtracking – Ausführung auf dem Enhanced Suffix Array und Abschätzung des Suchaufwands”. Bachelor thesis. Technische Universität München, 2012. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2012-bayer.pdf.

[BH05] S. J. Bedathur and J. R. Haritsa. “Search-Optimized Suffix-Tree Storage for Biological Applications”. In: *12th International Conference on High Performance Computing (HiPC’05)*. Volume 3769. LNCS. Goa, India: Springer, 2005, pages 29–39. http://dx.doi.org/10.1007/11602569_8.

[Beh+09] A. Behm, S. Ji, C. Li, and J. Lu. “Space-Constrained Gram-Based Indexing for Efficient Approximate String Search”. In: *24th International Conference on Data Engineering (ICDE’09)*. Shanghai, China: IEEE Computer Society, 2009, pages 604–615. http://dx.doi.org/10.1109/ICDE.2009.32.

[Beh+10] A. Behm, R. Vernica, S. Alsubaiee, S. Ji, J. Lu, L. Jin, Y. Lu, and C. Li. *UCI Flamingo Package*. Website, University of California, Irvine, School of Information and Computer Sciences. 2010. http://flamingo.ics.uci.edu/releases/4.1/ (visited on 2015-04-03).

[BS98] J. Bentley and B. Sedgewick. “Ternary Search Trees”. In: *Dr. Dobb’s Journal* 23.4 (1998), pages 20–25. http://www.drdobbs.com/database/ternary-search-trees/184410528 (visited on 2014-11-06).

[Ber86] J. Berstel. “Fibonacci Words – A Survey”. In: *The Book of L*. Springer, 1986, pages 13–27. http://dx.doi.org/10.1007/978-3-642-95486-3_2.

[Bin+13] T. Bingmann, J. Fischer, and V. Osipov. “Inducing Suffix and Lcp Arrays in External Memory”. In: *15th Meeting on Algorithm Engineering and Experiments (ALENEX’13)*. New Orleans, LA, USA: SIAM, 2013, pages 88–102. http://dx.doi.org/10.1137/1.9781611972931.8. (Visited on 2014-11-06).

[Boc+07] T. Bocek, E. Hunt, and B. Stiller. *Fast Similarity Search in Large Dictionaries*. Technical report. University of Zurich, 2007. http://fastss.csg.uzh.ch/ifi-2007.02.pdf (visited on 2014-11-06).

[Boo14] Boost. *Boost C++ libraries*. Website. 2014. http://www.boost.org/ (visited on 2014-11-06).

[Boy11] L. Boytsov. “Indexing Methods for Approximate Dictionary Searching: Comparative Analysis”. In: *Journal of Experimental Algorithmics* 16 (2011), 1.1:1.1–1.1:1.91. http://dx.doi.org/10.1145/1963190.1963191.

[Bra05] L. K. Branting. “Name Matching in Law Enforcement and CounterTerrorism”. In: *ICAIL Workshop on Data Mining, Information Extraction, and Evidentiary Reasoning for Law Enforcement and Counter-Terrorism*. Bologna, Italy, 2005, pages 28–31.

[BF06] G. S. Brodal and R. Fagerberg. “Cache-oblivious string dictionaries”. In: *17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA’06)*. Miami, Florida: ACM, 2006, pages 581–590. http://dx.doi.org/10.1145/1109557.1109621.

[BW94] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Technical report 124. Digital Equipment Corporation, 1994. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf (visited on 2014-11-06).

[Can12] Canonical Ltd. *Libcolumbus – A small, fast, error tolerant matcher*. Website. 2012. https://launchpad.net/libcolumbus (visited on 2015-04-03).

[Cao+05] X. Cao, S. C. Li, and A. K. Tung. "Indexing DNA Sequences using q-Grams". In: *10th International Conference on Database Systems for Advanced Applications (DASFAA'05)*. Volume 3453. LNCS. Beijing, China: Springer, 2005, pages 4–16. http://dx.doi.org/10.1007/11408079_4.

[Cha+06a] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. "A Linear Size Index for Approximate Pattern Matching". In: *17th Annual Symposium on Combinatorial Pattern Matching (CPM'06)*. Volume 4009. LNCS. Barcelona, Spain: Springer, 2006, pages 49–59. http://dx.doi.org/10.1007/11780441_6.

[Cha+06b] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. "Compressed Indexes for Approximate String Matching". In: *14th Annual European Symposium on Algorithms (ESA'06)*. Volume 4168. LNCS. Zurich, Switzerland: Springer, 2006, pages 208–219. http://dx.doi.org/10.1007/11841036_21.

[Cha06] S. Chapman. *SimMetrics – String similarity metrics for information integration*. Website, University of Sheffield. 2006. http://sourceforge.net/projects/simmetrics/ (visited on 2015-04-03).

[CN02] E. Chavez and G. Navarro. "A Metric Index for Approximate String Matching". In: *5th Latin American Symposium on Theoretical Informatics (LATIN'02)*. Volume 2286. LNCS. Cancun, Mexico: Springer, 2002, pages 181–195. http://dx.doi.org/10.1007/3-540-45995-2_20.

[Che+03] L.-L. Cheng, D. W.-L. Cheung, and S.-M. Yiu. "Approximate String Matching in DNA Sequences". In: *8th International Conference on Database Systems for Advanced Applications (DASFAA'03)*. Kyoto, Japan: IEEE Computer Society, 2003, pages 303–310. http://dx.doi.org/10.1109/DASFAA.2003.1192395.

[Cho+09] B. Chor, D. Horn, N. Goldman, Y. Levy, and T. Massingham. "Genomic DNA k-mer spectra: models and modalities". In: *Genome Biology* 10.10 (2009), R108. http://dx.doi.org/10.1186/gb-2009-10-10-r108.

[Cla08] F. Claude. *libcds – Compact Data Structures Library*. Website. 2008. https://github.com/fclaude/libcds (visited on 2015-04-03).

[Cla13] F. Claude. *libcds2 – A Compressed Data Structure Library*. Website. 2013. https://github.com/fclaude/libcds2 (visited on 2015-04-03).

[CS03] R. Clifford and M. Sergot. "Distributed and Paged Suffix Trees for Large Genetic Databases". In: *14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*. Volume 2676. LNCS. Morelia, Michoacán, Mexico: Springer, 2003, pages 70–82. http://dx.doi.org/10.1007/3-540-44888-8_6.

[CN10] R. Cánovas and G. Navarro. "Practical Compressed Suffix Trees". In: *9th International Symposium on Experimental Algorithms (SEA'10)*. Volume 6049. LNCS. Ischia Island, Italy: Springer, 2010, pages 94–105. http://dx.doi.org/10.1007/978-3-642-13193-6_9.

[Cob95] A. L. Cobbs. "Fast approximate matching using suffix trees". In: *6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*. Volume 937. LNCS. Espoo, Finland: Springer, 1995, pages 41–54. http://dx.doi.org/10.1007/3-540-60044-2_33.

[CO06] L. P. Coelho and A. L. Oliveira. “Dotted Suffix Trees – A Structure for Approximate Text Indexing”. In: *13th International Conference on String Processing and Information Retrieval (SPIRE’06)*. Volume 4209. LNCS. Glasgow, UK: Springer, 2006, pages 329–336. http://dx.doi.org/10.1007/11880561_27.

[Coh+03] W. W. Cohen, P. Ravikumar, S. Fienberg, and K. Rivard. *SecondString*. Website, Center for Automated Learning and Discovery, Carnegie Mellon University. 2003. http://secondstring.sourceforge.net/ (visited on 2015-04-03).

[Col+04] R. Cole, L.-A. Gottlieb, and M. Lewenstein. “Dictionary matching and indexing with errors and don’t cares”. In: *36th Annual ACM Symposium on Theory of Computing (STOC’04)*. Chicago, IL, USA: ACM, 2004, pages 91–100. http://dx.doi.org/10.1145/1007352.1007374.

[Col09] L. Collin. *XZ Utils – The next generation of LZMA Utils*. Website. 2009. http://tukaani.org/xz/ (visited on 2015-04-02).

[Con10] T. . G. P. Consortium. “A map of human genome variation from population-scale sequencing”. In: *Nature* 467 (2010), pages 1061–1073. http://dx.doi.org/10.1038/nature09534.

[CB84] A. Cornish-Bowden. “Nomenclature for incompletely specified bases in nucleic acid sequences”. In: *Nucleic Acids Research* 13.9 (1984), pages 3021–3030. http://dx.doi.org/10.1093/nar/13.9.3021.

[Dar+04] A. C. E. Darling, B. Mau, F. R. Blattner, and N. T. Perna. “Mauve: Multiple Alignment of Conserved Genomic Sequence With Rearrangements”. In: *Genome Res.* 14.7 (2004), pages 1394–1403. http://dx.doi.org/10.1101/gr.2289704.

[Dau10] A. Dau. “Analyse der Struktur und statistischer Eigenschaften von Texten und Erzeugung zufälliger Texte”. Bachelor thesis. Technische Universität München, 2010. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2010-dau.pdf.

[DK11a] A. Dau and J. Krugel. “tt-analyze and tt-generate: Tools to Analyze and Generate Sequences with Trained Statistical Properties”. In: *German Conference on Bioinformatics (GCB’11)*. Freising, Germany, 2011.

[DK11b] A. Dau and J. Krugel. *tt-analyze and tt-generate: Tools to Analyze and Generate Sequences with Trained Statistical Properties*. Technical report TUM-I1119. Technische Universität München, 2011. http://mediatum.ub.tum.de/doc/1097549/.

[Deh+03] M. Dehnert, W. E. Helm, and M.-T. Hütt. “A discrete autoregressive process as a model for short-range correlations in DNA sequences”. In: *Physica A: Statistical Mechanics and its Applications* 327.3–4 (2003), pages 535–553. http://dx.doi.org/10.1016/S0378-4371(03)00399-6.

[Den+13] D. Deng, G. Li, J. Feng, and W.-S. Li. “Top-k string similarity search with edit-distance constraints”. In: *29th IEEE International Conference on Data Engineering (ICDE’13)*. Brisbane, Australia: IEEE Computer Society, 2013, pages 925–936. http://dx.doi.org/10.1109/ICDE.2013.6544886.

[Deu96] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. Request for Comments 1951. 1996. https://tools.ietf.org/html/rfc1951 (visited on 2015-04-02).

[Dha+12] J. Dhaliwal, S. J. Puglisi, and A. Turpin. “Trends in Suffix Sorting: A Survey of Low Memory Algorithms”. In: *35th Australasian Computer Science Conference (ACSC’12)*. Volume 122. CRPIT. Melbourne, Australia: Australian Computer Society, 2012, pages 91–98. http://crpit.com/abstracts/CRPITV122Dhaliwal.html (visited on 2014-11-06).

[Dix+07] T. I. Dix, D. R. Powell, L. Allison, J. Bernal, S. Jaeger, and L. Stern. “Comparative analysis of long DNA sequences by per element information content using different contexts”. In: *BMC Bioinformatics* 8.Suppl 2 (2007), S10. http://dx.doi.org/10.1186/1471-2105-8-S2-S10.

[Dör+08] A. Döring, D. Weese, T. Rausch, and K. Reinert. “SeqAn – An efficient, generic C++ library for sequence analysis”. In: *BMC Bioinformatics* 9.11 (2008). Website, pages 1471–2105. http://dx.doi.org/10.1186/1471-2105-9-11. http://www.seqan.de/ (visited on 2015-04-02).

[Dun94] T. Dunning. *Statistical Identification of Language*. Technical report. New Mexico State University, 1994. http://ucrel.lancs.ac.uk/papers/lingdet.ps (visited on 2014-11-06).

[Edg+09] R. C. Edgar, G. Asimenos, S. Batzoglou, and A. Sidow. *Evolver*. Website. 2009. http://www.drive5.com/evolver (visited on 2014-11-06).

[EMG01] N. El-Mogharbel and J. A. M. Graves. “Encyclopedia of Life Sciences”. In: edited by D. N. Cooper. John Wiley & Sons, Ltd, 2001. Chapter X and Y Chromosomes: Homologous Regions, a0005793. ISBN: 9780470015902. http://dx.doi.org/10.1002/9780470015902.a0005793.pub2.

[Eli75] P. Elias. “Universal codeword sets and representations of the integers”. In: *IEEE Transactions on Information Theory* 21.2 (1975), pages 194–203. http://dx.doi.org/10.1109/TIT.1975.1055349.

[Far97] M. Farach. “Optimal Suffix Tree Construction with Large Alphabets”. In: *38th Annual Symposium on Foundations of Computer Science (FOCS’97)*. Miami Beach, FL, USA: IEEE Computer Society, 1997, pages 137–143. http://dx.doi.org/10.1109/SFCS.1997.646102.

[Fat+05] M. Fatemi, M. M. Pao, S. Jeong, E. N. Gal-Yam, G. Egger, D. J. Weisenberger, and P. A. Jones. “Footprinting of mammalian promoters: use of a CpG DNA methyltransferase revealing nucleosome positions at a single molecule level”. In: *Nucleic Acids Res.* 33.20 (2005), e176. http://dx.doi.org/10.1093/nar/gni180.

[Fer12] M. P. Ferguson. “FEMTO: Fast Search of Large Sequence Collections”. In: *23rd Annual Symposium on Combinatorial Pattern Matching (CPM’12)*. Volume 7354. LNCS. Helsinki, Finland: Springer, 2012, pages 208–219. http://dx.doi.org/10.1007/978-3-642-31265-6_17.

[FF07] P. Ferragina and J. Fischer. “Suffix Arrays on Words”. In: *18th Annual Symposium on Combinatorial Pattern Matching (CPM’07)*. Volume 4580. LNCS. London, Ontario, Canada: Springer, 2007, pages 328–339. http://dx.doi.org/10.1007/978-3-540-73437-6_33.

[FG99] P. Ferragina and R. Grossi. “The string B-tree: a new data structure for string search in external memory and its applications”. In: *Journal of the ACM* 46.2 (1999), pages 236–280. http://dx.doi.org/10.1145/301970.301973.

[FM00] P. Ferragina and G. Manzini. “Opportunistic data structures with applications”. In: *41st Annual Symposium on Foundations of Computer Science (FOCS’00)*. Redondo Beach, CA, USA: IEEE Computer Society, 2000, page 390. http://dx.doi.org/10.1109/SFCS.2000.892127.

[FM05] P. Ferragina and G. Manzini. “Indexing compressed text”. In: *Journal of the ACM* 52.4 (2005), pages 552–581. http://dx.doi.org/10.1145/1082036.1082039.

[FN05] P. Ferragina and G. Navarro. *The Pizza & Chili Corpus – Compressed Indexes and their Testbeds*. Website, University of Pisa / University of Chile. http://pizzachili.dcc.uchile.cl/. 2005. http://pizzachili.di.unipi.it/ (visited on 2015-04-03).

[Fer+04] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. "An Alphabet-Friendly FM-index". In: *11th International Conference on String Processing and Information Retrieval (SPIRE'04)*. Volume 3246. LNCS. Padova, Italy: Springer, 2004, pages 150–160. http://dx.doi.org/10.1007/b100941.

[Fer+09a] P. Ferragina, R. González, G. Navarro, and R. Venturini. "Compressed text indexes: from theory to practice". In: *Journal of Experimental Algorithmics* 13 (2009), pages 1.12–1.31. http://dx.doi.org/10.1145/1412228.1455268.

[Fer+09b] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. "Compressing and indexing labeled trees, with applications". In: *Journal of the ACM* 57.1 (2009), pages 1–33. http://dx.doi.org/10.1145/1613676.1613680.

[Fis11] J. Fischer. "Inducing the LCP-Array". In: *12th International Symposium on Algorithms and Data Structures (WADS'11)*. Volume 6844. LNCS. Springer, 2011, pages 374–385. http://dx.doi.org/10.1007/978-3-642-22300-6_32.

[Fis+06] J. Fischer, V. Heun, and S. Kramer. "Optimal String Mining Under Frequency Constraints". In: *10th European Conference on Principle and Practice of Knowledge Discovery in Databases (PKDD'06)*. Berlin, Germany: Springer, 2006, pages 139–150. http://dx.doi.org/10.1007/11871637_17.

[Fis+08] J. Fischer, V. Mäkinen, and G. Navarro. "An(other) Entropy-Bounded Compressed Suffix Tree". In: *19th Annual Symposium on Combinatorial Pattern Matching (CPM'08)*. Volume 5029. LNCS. Pisa, Italy: Springer, 2008, pages 152–165. http://dx.doi.org/10.1007/978-3-540-69068-9_16.

[Fis+09] J. Fischer, V. Mäkinen, and G. Navarro. "Faster entropy-bounded compressed suffix trees". In: *Theoretical Computer Science* 410.51 (2009). Combinatorial Pattern Matching, pages 5354–5364. http://dx.doi.org/10.1016/j.tcs.2009.09.012.

[Fre60] E. Fredkin. "Trie memory". In: *Communications of the ACM* 3.9 (1960), pages 490–499. http://dx.doi.org/10.1145/367390.367400.

[Gag+11] T. Gagie, P. Gawrychowski, and S. Puglisi. "Faster Approximate Pattern Matching in Compressed Repetitive Texts". In: *22nd International Symposium on Algorithms and Computation (ISAAC'11)*. Volume 7074. LNCS. Yokohama, Japan: Springer, 2011, pages 653–662. http://dx.doi.org/10.1007/978-3-642-25591-5_67.

[Gie+03] R. Giegerich, S. Kurtz, and J. Stoye. "Efficient Implementation of Lazy Suffix Trees". In: *Software – Practice and Experience* 33.11 (2003), pages 1035–1049. http://dx.doi.org/10.1002/spe.535.

[Gog14] S. Gog. *sdsl – Succinct Data Structure Library*. Website, Universität Ulm. 2014. http://www.uni-ulm.de/in/theo/research/sdsl.html (visited on 2015-04-03).

[GF10] S. Gog and J. Fischer. "Advantages of Shared Data Structures for Sequences of Balanced Parentheses". In: *Data Compression Conference (DCC'10)*. Snowbird, UT, USA: IEEE Computer Society, 2010, pages 406–415. http://dx.doi.org/10.1109/DCC.2010.43.

[GO11] S. Gog and E. Ohlebusch. "Fast and Lightweight LCP-Array Construction Algorithms". In: *13th Workshop on Algorithm Engineering and Experiments (ALENEX'11)*. San Francisco, CA, USA: SIAM, 2011, pages 25–34. http://dx.doi.org/10.1137/1.9781611972917.3.

[GDR09] A. Gogol-Döring and K. Reinert. *Biological Sequence Analysis Using the SeqAn C++ Library*. Mathematical and Computational Biology Series. Chapman & Hall / CRC Press, 2009. ISBN: 9781420076233. http://www.seqan.de/ (visited on 2014-11-06).

[Gon+92] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. “New indices for text: PAT trees and PAT arrays”. In: *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992, pages 66–82. ISBN: 9780134638379. (Visited on 2014-11-06).

[Gro+00] I. Grosse, H. Herzel, S. V. Buldyrev, and H. E. Stanley. “Species independence of mutual information in coding and noncoding DNA”. In: *Physical Review E* 61.5 (2000), page 5624. http://dx.doi.org/10.1103/PhysRevE.61.5624.

[Gro11] R. Grossi. “A quick tour on suffix arrays and compressed suffix arrays”. In: *Theoretical Computer Science* 412.27 (2011), pages 2964–2973. http://dx.doi.org/10.1016/j.tcs.2010.12.036.

[GV00] R. Grossi and J. S. Vitter. “Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching (extended abstract)”. In: *32nd Annual ACM Symposium on Theory of Computing (STOC’00)*. Portland, OR, USA: ACM, 2000, pages 397–406. http://dx.doi.org/10.1145/335305.335351.

[GV05] R. Grossi and J. S. Vitter. “Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching”. In: *SIAM Journal on Computing* 35.2 (2005), pages 378–407. http://dx.doi.org/10.1137/S0097539702402354.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. ISBN: 9780521585194.

[HL09] M. Hadjieleftheriou and C. Li. “Efficient approximate search on string collections”. In: *Proceedings of the VLDB Endowment* 2.2 (2009). Tutorial, pages 1660–1661. http://www.vldb.org/pvldb/2/vldb09-tutorial4.pdf (visited on 2014-11-06).

[Hal+07] M. Halachev, N. Shiri, and A. Thamildurai. “Efficient and Scalable Indexing Techniques for Biological Sequence Data”. In: *1st International Conference on Bioinformatics Research and Development (BIRD’07)*. Volume 4414. LNCS. Berlin, Germany: Springer, 2007, pages 464–479. http://dx.doi.org/10.1007/978-3-540-71233-6_36.

[Ham50] R. W. Hamming. “Error detecting and error correcting codes”. In: *Bell System Technical Journal* 29.2 (1950), pages 147–160. http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x.

[Han+14] H. Hanada, M. Kudo, and A. Nakamura. “Average-case linear-time similar substring searching by the q-gram distance”. In: *Theoretical Computer Science* 530 (2014), pages 23–41. http://dx.doi.org/10.1016/j.tcs.2014.02.022.

[Har71] M. S. Hart. *Project Gutenberg*. Website. 1971. https://www.gutenberg.org/ (visited on 2014-11-06).

[Hei+02] S. Heinz, J. Zobel, and H. E. Williams. “Burst tries: A fast, efficient data structure for string keys”. In: *ACM Transactions on Information Systems* 20.2 (2002), pages 192–223. http://dx.doi.org/10.1145/506309.506312.

[HG95] H. Herzel and I. Große. “Measuring correlations in symbol sequences”. In: *Physica A: Statistical and Theoretical Physics* 216.4 (1995), pages 518–542. http://dx.doi.org/10.1016/0378-4371(95)00104-F.

[Her+94] H. Herzel, W. Ebeling, and A. O. Schmitt. “Entropies of biosequences: The role of repeats”. In: *Physical Review E* 50.6 (1994), pages 5061–5071. http://dx.doi.org/10.1103/PhysRevE.50.5061.

[Hie13] J. Hietaniemi. *String::Approx – Perl extension for approximate matching (fuzzy matching)*. Website. 2013. http://search.cpan.org/dist/String-Approx/ (visited on 2015-04-03).

[Hig05] C. de la Higuera. “A bibliographical study of grammatical inference”. In: *Pattern Recognition* 38.9 (2005), pages 1332–1348. http://dx.doi.org/10.1016/j.patcog.2005.01.003.

[Hol+03] D. Holste, I. Grosse, S. Beirer, P. Schieg, and H. Herzel. “Repeats and correlations in human DNA sequences”. In: *Physical Review E* 67.6 (2003), page 061913. http://dx.doi.org/10.1103/PhysRevE.67.061913.

[Hol10] M. Holtgrewe. *Mason – A Read Simulator for Second Generation Sequencing Data*. Technical report B-10-06. Freie Universität Berlin, 2010. http://www.seqan.de/projects/mason/ (visited on 2014-11-06).

[Hon+03] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. “Constructing Compressed Suffix Arrays with Large Alphabets”. In: *14th International Symposium on Algorithms and Computation 2003 (ISAAC’03)*. Volume 2906. LNCS. Kyoto, Japan: Springer, 2003, pages 240–249. http://dx.doi.org/10.1007/978-3-540-24587-2_26.

[Hon+04] W.-K. Hon, T.-W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. “Practical Aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences”. In: *6th Workshop on Algorithm Engineering and Experiments and the 1st Workshop on Analytic Algorithmics and Combinatorics (ALENEX’04)*. New Orleans, LA, USA: SIAM, 2004, pages 31–38. http://www.siam.org/meetings/alenex04/abstacts/WHon.pdf (visited on 2014-11-06).

[Hon+07a] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. “A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays”. In: *Algorithmica* 48.1 (1 2007), pages 23–36. http://dx.doi.org/10.1007/s00453-006-1228-8.

[Hon+07b] W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. “Cache-Oblivious Index for Approximate String Matching”. In: *18th Annual Symposium on Combinatorial Pattern Matching (CPM’07)*. Volume 4580. LNCS. London, Ontario, Canada: Springer, 2007, pages 40–51. http://dx.doi.org/10.1007/978-3-540-73437-6_7.

[Hon+10] W.-K. Hon, R. Shah, and J. Vitter. “Compression, Indexing, and Retrieval for Massive String Data”. In: *21st Annual Symposium on Combinatorial Pattern Matching (CPM’10)*. Volume 6129. LNCS. Lille, France: Springer, 2010, pages 260–274. http://dx.doi.org/10.1007/978-3-642-13509-5_24.

[Hon+11] W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. “Cache-oblivious index for approximate string matching”. In: *Theoretical Computer Science* 412.29 (2011), pages 3579–3588. http://dx.doi.org/10.1016/j.tcs.2011.03.004.

[Hor80] R. N. Horspool. “Practical fast searching in strings”. In: *Software – Practice and Experience* 10.6 (1980), pages 501–506. http://dx.doi.org/10.1002/spe.4380100608.

[HD06] M.-T. Hütt and M. Dehnert. *Methoden der Bioinformatik*. Springer, 2006. http://dx.doi.org/10.1007/3-540-32954-4.

[Hun+01] E. Hunt, M. P. Atkinson, and R. W. Irving. “A Database Index to Large Biological Sequences”. In: *27th International Conference on Very Large Data Bases (VLDB’01)*. Roma, Italy: Morgan Kaufmann Publishers Inc., 2001, pages 139–148. http://www.vldb.org/conf/2001/P139.pdf (visited on 2014-11-06).

[Hur+06] C.-G. Hur, S. Kim, C. H. Kim, S. H. Yoon, Y.-H. In, C. Kim, and H. G. Cho. “FASIM: Fragments assembly simulation using biased-sampling model and assembly simulation for microbial genome shotgun sequencing”. In: *Journal of Microbiology and Biotechnology* 16.5 (2006), pages 683–688. http://www.jmb.or.kr/submission/Journal/016/JMB016-05-06.pdf.

[Huy+04] T. N. D. Huynh, W.-K. Hon, T.-W. Lam, and W.-K. Sung. “Approximate String Matching using Compressed Suffix Arrays”. In: *15th Annual Symposium on Combinatorial Pattern Matching (CPM’04)*. Volume 3109. LNCS. Istanbul, Turkey: Springer, 2004, pages 434–444. http://dx.doi.org/10.1007/b98377.

[Huy+06] T. N. Huynh, W.-K. Hon, T.-W. Lam, and W.-K. Sung. “Approximate string matching using compressed suffix arrays”. In: *Theoretical Computer Science* 352.1-3 (2006), pages 240–249. http://dx.doi.org/10.1016/j.tcs.2005.11.022.

[Hyy01] H. Hyyrö. *Explaining and extending the bit-parallel approximate string matching algorithm of Myers*. Technical report A-2001-10. Department of Computer and Information Sciences, University of Tampere, 2001. http://www.cs.uta.fi/~helmu/pubs/A2001-10.pdf (visited on 2014-11-06).

[HN03] H. Hyyrö and G. Navarro. “A Practical Index for Genome Searching”. In: *10th International Symposium on String Processing and Information Retrieval (SPIRE’03)*. Volume 2857. LNCS. Manaus, Brazil: Springer, 2003, pages 341–349. http://dx.doi.org/10.1007/b14038.

[Ili+97] C. S. Iliopoulos, D. Moore, and W. F. Smyth. “A characterization of the squares in a Fibonacci string”. In: *Theoretical Computer Science* 172.1–2 (1997), pages 281–291. http://dx.doi.org/10.1016/S0304-3975(96)00141-7.

[Int04] International Human Genome Sequencing Consortium. “Finishing the euchromatic sequence of the human genome”. In: *Nature* 431 (2004), pages 931–945. http://dx.doi.org/10.1038/nature03001.

[JL83] P. A. Jacobs and P. A. W. Lewis. “Stationary Discrete Autoregressive-Moving Average Time Series Generated By Mixtures”. In: *Journal of Time Series Analysis* 4.1 (1983), pages 19–36. http://dx.doi.org/10.1111/j.1467-9892.1983.tb00354.x.

[Jap04] R. Japp. *The Top-Compressed Suffix Tree: A Disk-Resident Index for Large Sequences*. Technical report TR-2004-165. University of Glasgow, Department of Computing Science, 2004. http://www.dcs.gla.ac.uk/publications/PAPERS/7832/Jap04_TCST_TechRep.pdf (visited on 2014-11-06).

[JU91] P. Jokinen and E. Ukkonen. “Two Algorithms for Approximate String Matching in Static Texts”. In: *16th International Symposium on Mathematical Foundations of Computer Science (MFCS’91)*. Volume 520. LNCS. Kazimierz Dolny, Poland: Springer, 1991, pages 240–248. http://dx.doi.org/10.1007/3-540-54345-7_67.

[Jok+96] P. Jokinen, J. Tarhio, and E. Ukkonen. “A comparison of approximate string matching algorithms”. In: *Software – Practice and Experience* 26.12 (1996), pages 1439–1458. http://dx.doi.org/10.1002/(SICI)1097-024X(199612)26:12<1439::AID-SPE71>3.0.CO;2-1.

[Kap+09] O. Kapah, G. M. Landau, A. Levy, and N. Oz. “Interchange rearrangement: The element-cost model”. In: *Theoretical Computer Science* 410.43 (2009), pages 4315–4326. http://dx.doi.org/10.1016/j.tcs.2009.07.013.

[Kas+01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. “Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications”. In: *12th Annual Symposium on Combinatorial Pattern Matching (CPM’06)*. Volume 2089. LNCS. Jerusalem, Israel: Springer, 2001, pages 181–192. http://dx.doi.org/10.1007/3-540-48194-X_17.

[Kim+05] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. “n-Gram/2L: A Space and Time Efficient Two-Level n-Gram Inverted Index Structure”. In: *31st International Conference on Very Large Data Bases (VLDB’05)*. Trondheim, Norway: ACM, 2005, pages 325–336. http://www.vldb.org/conf/2005/papers/p325-kim.pdf (visited on 2014-11-06).

[Kim+07] M.-S. Kim, K.-Y. Whang, and J.-G. Lee. “n-gram/2L-approximation: A two-level n-gram inverted index structure for approximate string matching”. In: *Computer Systems Science & Engineering* 22.6 (2007), pages 26–40. http://infolab.dgist.ac.kr/~mskim/papers/CSSE07.pdf (visited on 2014-11-06).

[Kim+08] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. “Structural optimization of a full-text *n*-gram index using relational normalization”. In: *The VLDB Journal* 17 (6 2008), pages 1485–1507. http://dx.doi.org/10.1007/s00778-007-0082-x.

[KS13] Y. Kim and K. Shim. “Efficient Top-k Algorithms for Approximate Substring Matching”. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD’13)*. New York, NY, USA: ACM, 2013, pages 385–396. http://dx.doi.org/10.1145/2463676.2465324.

[Kim+10] Y. Kim, K.-G. Woo, H. Park, and K. Shim. “Efficient processing of substring match queries with inverted q-gram indexes”. In: *26th International Conference on Data Engineering (ICDE’10)*. Long Beach, CA: IEEE Computer Society, 2010, pages 721–732. http://dx.doi.org/10.1109/ICDE.2010.5447866.

[Kly09] R. S. Klyujkov. *PATL – Practical Algorithm Template Library*. Website. 2009. https://code.google.com/p/patl/ (visited on 2015-04-03).

[Knu98] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. 2nd. Volume 3. Addison-Wesley, 1998. ISBN: 9780201896855.

[KS03] J. Kärkkäinen and P. Sanders. “Simple Linear Work Suffix Array Construction”. In: *30th International Colloquium on Automata, Languages and Programming (ICALP’03)*. Volume 2719. LNCS. Eindhoven, The Netherlands: Springer, 2003, page 187. http://dx.doi.org/10.1007/3-540-45061-0_73.

[Kär+06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. “Linear work suffix array construction”. In: *Journal of the ACM* 53.6 (2006), pages 918–936. http://dx.doi.org/10.1145/1217856.1217858.

[Kru08] J. Krugel. “Suche von ähnlichen Datensätzen unter Echtzeitbedingungen”. Diploma thesis. Freie Universität Berlin, 2008. http://www.mi.fu-berlin.de/en/inf/groups/ag-db/jobs_and_theses/finished_thesis/KrugelDipl.pdf (visited on 2014-11-06).

[Kuk92] K. Kukich. “Techniques for automatically correcting words in text”. In: *ACM Computing Surveys* 24.4 (1992), pages 377–439. http://dx.doi.org/10.1145/146370.146380.

[Kur99] S. Kurtz. “Reducing the space requirement of suffix trees”. In: *Software – Practice and Experience* 29.13 (1999), pages 1149–1171. http://dx.doi.org/10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

[Lam+08a] T.-W. Lam, W.-K. Sung, S.-L. Tam, C.-K. Wong, and S.-M. Yiu. “Compressed indexing and local alignment of DNA”. In: *Bioinformatics* 24.6 (2008), pages 791–797. http://dx.doi.org/10.1093/bioinformatics/btn032.

[Lam+08b] T.-W. Lam, W.-K. Sung, and S.-S. Wong. “Improved Approximate String Matching Using Compressed Suffix Data Structures”. In: *Algorithmica* 51.3 (2008), pages 298–314. http://dx.doi.org/10.1007/s00453-007-9104-8.

[LS12] B. Langmead and S. L. Salzberg. “Fast gapped-read alignment with Bowtie 2”. In: *Nature Methods* 9 (2012), pages 357–359. http://dx.doi.org/10.1038/nmeth.1923.

[Lan+09] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”. In: *Genome Biology* 10.3 (2009), R25. http://dx.doi.org/10.1186/gb-2009-10-3-r25. http://bowtie-bio.sourceforge.net/ (visited on 2014-11-06).

[LS07] N. J. Larsson and K. Sadakane. “Faster suffix sorting”. In: *Theoretical Computer Science* 387.3 (2007), pages 258–272. http://dx.doi.org/10.1016/j.tcs.2007.07.017.

[Lau+92] G. Lauc, I. Ilic, and M. Heffer-Lauc. “Entropies of coding and noncoding sequences of DNA and proteins”. In: *Biophys. Chem.* 42.1 (1992), pages 7–11. http://dx.doi.org/10.1016/0301-4622(92)80002-M.

[Lau09] V. Laurikari. *TRE – The free and portable approximate regex matching library*. 2009. http://laurikari.net/tre/ (visited on 2015-04-03).

[Li+07] C. Li, B. Wang, and X. Yang. “VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams”. In: *33rd International Conference on Very Large Data Bases (VLDB’07)*. Vienna, Austria: ACM, 2007, pages 303–314. http://www.vldb.org/conf/2007/papers/research/p303-li.pdf (visited on 2014-11-06).

[Li+08] C. Li, J. Lu, and Y. Lu. “Efficient Merging and Filtering Algorithms for Approximate String Searches”. In: *24th International Conference on Data Engineering (ICDE’08)*. Cancún, México: IEEE Computer Society, 2008, pages 257 –266. http://dx.doi.org/10.1109/ICDE.2008.4497434.

[LK92] W. Li and K. Kaneko. “Long-Range Correlation and Partial $1/f^{\alpha}$ Spectrum in a Noncoding DNA Sequence”. In: *EPL – Europhysics Letters* 17.7 (1992), page 655. http://dx.doi.org/10.1209/0295-5075/17/7/014.

[Lot97] M. Lothaire. *Combinatorics on Words*. Cambridge University Press, 1997. ISBN: 9780521599245.

[Lyo98] M. F. Lyon. “X-chromosome inactivation: a repeat hypothesis”. In: *Cytogenetics and Cell Genetics* 80 (1998), pages 133–7.

[Maa04] M. G. Maaß. “Average-Case Analysis of Approximate Trie Search”. In: *15th Annual Symposium on Combinatorial Pattern Matching (CPM’04)*. Volume 3109. LNCS. Istanbul, Turkey: Springer, 2004, pages 472–483. http://dx.doi.org/10.1007/b98377.

[MN05a] M. G. Maaß and J. Nowak. “Text Indexing with Errors”. In: *16th Annual Symposium on Combinatorial Pattern Matching (CPM’05)*. Volume 3537. LNCS. Jeju Island, Korea: Springer, 2005, pages 21–32. http://dx.doi.org/10.1007/11496656_3.

[Mad13] R. Madden. *stringmetric – String metrics and phonetic algorithms for Scala*. Website. 2013. https://github.com/rockymadden/stringmetric (visited on 2015-04-03).

[Mad01] J. Maddock. “Regular Expressions in C++”. In: *Dr. Dobb’s Journal* 26.10 (2001), pages 21–26. http://www.boost.org/doc/libs/release/libs/regex/ (visited on 2015-04-03).

[MM93] U. Manber and E. W. Myers. “Suffix Arrays: A New Method for On-Line String Searches”. In: *SIAM Journal on Computing* 22.5 (1993), pages 935–948. http://dx.doi.org/10.1137/0222058.

[MM90] U. Manber and G. Myers. “Suffix Arrays: A New Method for On-Line String Searches”. In: *1st Annual Symposium on Discrete Algorithms (SODA’90)*. San Francisco, CA, USA: SIAM, 1990, pages 319–327. http://dl.acm.org/citation.cfm?id=320176.320218 (visited on 2014-11-06).

[MM07] H. Mangs and B. Morris. “The Human Pseudoautosomal Region (PAR): Origin, Function and Future”. In: *Current Genomics* 8 (2007), pages 129–136. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2435358/ (visited on 2015-04-02).

[Man+11] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. “ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings”. In: *Proceedings of the VLDB Endowment* 5.1 (2011), pages 49–60. ISSN: 2150-8097. http://www.vldb.org/pvldb/vol5/p049_essammansour_vldb2012.pdf (visited on 2014-11-06).

[Man+94] R. N. Mantegna, S. V. Buldyrev, A. L. Goldberger, S. Havlin, C. K. Peng, M. Simons, and H. E. Stanley. “Linguistic Features of Noncoding DNA Sequences”. In: *Physical Review Letters* 73.23 (1994), pages 3169–3172. http://dx.doi.org/10.1103/PhysRevLett.73.3169.

[Man01] G. Manzini. “An analysis of the Burrows-Wheeler transform”. In: *Journal of the ACM* 48.3 (2001), pages 407–430. http://dx.doi.org/10.1145/382780.382782.

[Man04] G. Manzini. “Two Space Saving Tricks for Linear Time LCP Array Computation”. In: *9th Scandinavian Workshop on Algorithm Theory (SWAT’04)*. Volume 3111. LNCS. Humlebaek, Denmark: Springer, 2004, pages 372–383. http://dx.doi.org/10.1007/978-3-540-27810-8_32.

[MK11] G. Marçais and C. Kingsford. “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”. In: *Bioinformatics* 27.6 (2011), pages 764–770. http://dx.doi.org/10.1093/bioinformatics/btr011.

[McC76] E. M. McCreight. “A Space-Economical Suffix Tree Construction Algorithm”. In: *Jounal of the ACM* 23.2 (1976), pages 262–272. http://dx.doi.org/10.1145/321941.321946.

[MN99] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Pressh, 1999. ISBN: 9780521563291. http://www.mpi-inf.mpg.de/~mehlhorn/LEDAbook.html (visited on 2014-11-06).

[Mer12] J. Merkle. “Speicherplatzeffiziente q-Gramm-Indexe”. Bachelor thesis. Technische Universität München, 2012. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2012-merkle.pdf.

[Müh08] A. Mühling. “Approximate Pattern Matching”. Master thesis. Technische Universität München, 2008.

[MS04] S. Mihov and K. U. Schulz. “Fast Approximate Search in Large Dictionaries”. In: *Computational Linguistics* 30.4 (2004), pages 451–477. http://dx.doi.org/10.1162/0891201042544938.

[Min+14] D. Minkley, M. Whitney, S.-H. Lin, M. Barsky, C. Kelly, and C. Upton. “Suffix tree searcher: exploration of common substrings in large DNA sequence sets”. In: *BMC Research Notes* 7.1 (2014), page 466. http://dx.doi.org/10.1186/1756-0500-7-466.

[MN05b] V. Mäkinen and G. Navarro. “Succinct Suffix Arrays Based on Run-Length Encoding”. In: *16th Annual Symposium on Combinatorial Pattern Matching (CPM’05)*. Volume 3537. LNCS. Jeju Island, Korea: Springer, 2005, pages 45–56. http://dx.doi.org/10.1007/11496656_5.

[Mor08] Y. Mori. *libdivsufsort – A lightweight suffix-sorting library*. Website. 2008. https://code.google.com/p/libdivsufsort/ (visited on 2014-11-06).

[Mor68] D. R. Morrison. “PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric”. In: *Journal of the ACM* 15.4 (1968), pages 514–534. http://dx.doi.org/10.1145/321479.321481.

[Mye94] E. W. Myers. "A sublinear algorithm for approximate keyword searching". In: *Algorithmica* 12.4–5 (1994), pages 345–374. http://dx.doi.org/10.1007/BF01185432.

[Mye99a] G. Myers. "A dataset generator for whole genome shotgun sequencing". In: *7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99)*. Heidelberg, Germany: AAAI Press, 1999, pages 202–210. http://www.aaai.org/Papers/ISMB/1999/ISMB99-024.pdf (visited on 2014-11-06).

[Mye99b] G. Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM* 46.3 (1999), pages 395–415. http://dx.doi.org/10.1145/316542.316550.

[NVI14] NVIDIA Corporation. *NVBIO*. Website. 2014. http://nvlabs.github.io/nvbio/ (visited on 2015-04-03).

[Nat09] National Center for Biotechnology Information. *GenBank*. 2009. https://www.ncbi.nlm.nih.gov/genbank/.

[Nav01] G. Navarro. "A guided tour to approximate string matching". In: *ACM Computing Surveys* 33.1 (2001), pages 31–88. http://dx.doi.org/10.1145/375360.375365.

[Nav02] G. Navarro. "Indexing Text Using the Ziv-Lempel Trie". In: *9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*. Volume 2476. LNCS. Lisbon, Portugal: Springer, 2002, pages 325–336. http://dx.doi.org/10.1007/3-540-45735-6_28.

[Nav04] G. Navarro. "Indexing text using the Ziv-Lempel trie". In: *Journal of Discrete Algorithms* 2.1 (2004), pages 87–114. http://dx.doi.org/10.1016/S1570-8667(03)00066-2.

[Nav09] G. Navarro. "Implementing the LZ-index: Theory versus practice". In: *Journal of Experimental Algorithmics* 13 (2009), 2:1.2–2:1.49. http://dx.doi.org/10.1145/1412228.1412230.

[Nav11] G. Navarro. *Indexed Approximate String Matching*. Open Problem, International Workshop on Combinatorial Algorithms (IWOCA'11). 2011. http://www.iwoca.org/problems/Navarro2.pdf (visited on 2014-11-06).

[Nav12] G. Navarro. "Indexing Highly Repetitive Collections". In: *Combinatorial Algorithms*. Volume 7643. LNCS. Springer, 2012, pages 274–279. http://dx.doi.org/10.1007/978-3-642-35926-2_29.

[NBY98] G. Navarro and R. Baeza-Yates. "A practical *q*-gram index for text retrieval allowing errors". In: *CLEI Electronic Journal* 1.2 (1998), pages 31–88. http://www.clei.cl/cleiej/papers/v1i2p3.pdf (visited on 2014-11-06).

[NBY99] G. Navarro and R. Baeza-Yates. "Very fast and simple approximate string matching". In: *Information Processing Letters* 72.1-2 (1999), pages 65–70. http://dx.doi.org/10.1016/S0020-0190(99)00121-0.

[NBY00] G. Navarro and R. Baeza-Yates. "A Hybrid Indexing Method for Approximate String Matching". In: *Journal of Discrete Algorithms* 1.1 (2000). (Special issue on Matching Patterns), pages 205–239. http://www.dcc.uchile.cl/~gnavarro/ps/jda00.2.pdf (visited on 2014-11-06).

[NM07] G. Navarro and V. Mäkinen. "Compressed full-text indexes". In: *ACM Computing Surveys* 39.1 (2007), page 2. http://dx.doi.org/10.1145/1216370.1216372.

[NR00] G. Navarro and M. Raffinot. "Fast and Flexible String Matching by Combining Bit-parallelism and Suffix Automata". In: *Journal of Experimental Algorithmics* 5 (2000), pages 1–36. http://dx.doi.org/10.1145/351827.384246.

[NR02] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN: 9780521813075. http://www.dcc.uchile.cl/~gnavarro/FPMbook/ (visited on 2014-11-06).

[NS09] G. Navarro and L. Salmela. “Indexing Variable Length Substrings for Exact and Approximate Matching”. In: *16th International Symposium on String Processing and Information Retrieval (SPIRE’09)*. Volume 5721. LNCS. Saariselkä, Finland: Springer, 2009, pages 214–221. http://dx.doi.org/10.1007/978-3-642-03784-9_21.

[Nav+00] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. “Indexing Text with Approximate *q*-Grams”. In: *11th Annual Symposium on Combinatorial Pattern Matching (CPM’00)*. Volume 1848. LNCS. Montreal, Canada: Springer, 2000, pages 350–363. http://dx.doi.org/10.1007/3-540-45123-4_29.

[Nav+01] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. “Indexing Methods for Approximate String Matching”. In: *IEEE Data Engineering Bulletin* 24.4 (2001), pages 19–27. http://sites.computer.org/debull/A01DEC-CD.pdf (visited on 2014-11-06).

[Nav+05] G. Navarro, E. Sutinen, and J. Tarhio. “Indexing text with approximate *q*-grams”. In: *Journal of Discrete Algorithms* 3.2-4 (2005), pages 157–175. http://dx.doi.org/10.1016/j.jda.2004.08.003.

[NW70] S. B. Needleman and C. D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of Molecular Biology* 48.3 (1970), pages 443–453. http://dx.doi.org/10.1016/0022-2836(70)90057-4.

[Ohl13] E. Ohlebusch. *Bioinformatics Algorithms*. Bremen, Germany: Oldenbusch Verlag, 2013. ISBN: 9783000413162. http://www.uni-ulm.de/in/theo/m/ohlebusch/book-bioinformatics-algorithms.html (visited on 2014-11-06).

[Ohl+10] E. Ohlebusch, J. Fischer, and S. Gog. “CST++”. In: *17th International Symposium on String Processing and Information Retrieval (SPIRE’10)*. Volume 6393. LNCS. Los Cabos, México: Springer, 2010, pages 322–333. http://dx.doi.org/10.1007/978-3-642-16321-0_34.

[OT10] N. Okazaki and J. Tsujii. “Simple and Efficient Algorithm for Approximate Dictionary Matching”. In: *23rd Int. Conf. on Comput. Linguistics (Coling’10)*. http://www.chokkan.org/software/simstring/. Beijing, China: Tsinghua University Press, 2010, pages 851–859. http://www.aclweb.org/anthology/C10-1096 (visited on 2014-11-06).

[Pat+13] M. Patil, X. Cai, S. V. Thankachan, R. Shah, S.-J. Park, and D. Foltz. “Approximate string matching by position restricted alignment”. In: *Joint EDBT/ICDT 2013 Workshops (EDBT’13)*. Genoa, Italy: ACM, 2013, pages 384–391. http://dx.doi.org/10.1145/2457317.2457388.

[Pen+92] C. K. Peng, S. V. Buldyrev, A. L. Goldberger, S. Havlin, F. Sciortino, M. Simon, and H. E. Stanley. “Long-range correlations in nucleotide sequences”. In: *Nature* 356 (1992), pages 168–170. http://dx.doi.org/10.1038/356168a0.

[PZ07] B. Phoophakdee and M. J. Zaki. “Genome-scale disk-based suffix tree indexing”. In: *International Conference on Management of Data (SIGMOD’07)*. Beijing, China: ACM, 2007, pages 833–844. http://dx.doi.org/10.1145/1247480.1247572.

[PZ08] B. Phoophakdee and M. J. Zaki. “TRELLIS+: An Effective Approach for Indexing Genome-Scale Sequences Using Suffix Trees”. In: *Pacific Symposium on Biocomputing (PSB’08)*. Kohala Coast, Hawaii, USA: World Scientific, 2008, pages 90–101. http://psb.stanford.edu/psb-online/proceedings/psb08/phoophakdee.pdf (visited on 2014-11-06).

[Pop10] H. Poppe. “Approximative Suche in Textindizes”. Bachelor thesis. Technische Universität München, 2010. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2010-poppe.pdf.

[Pot+12] M. Potthast et al. “Overview of the 4th International Competition on Plagiarism Detection”. In: *Conference and Labs of the Evaluation Forum (CLEF’12), Evaluation Labs and Workshop, Online Working Notes*. Volume 1178. Rome, Italy, 2012. http://ceur-ws.org/Vol-1178/CLEF2012wn-PAN-PotthastEt2012.pdf (visited on 2015-04-02).

[Pug+06] S. J. Puglisi, W. Smyth, and A. Turpin. “Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory”. In: *13th International Conference on String Processing and Information Retrieval (SPIRE’06)*. Volume 4209. LNCS. Glasgow, UK: Springer, 2006, pages 122–133. http://dx.doi.org/10.1007/11880561_11.

[Pug+07] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. “A taxonomy of suffix array construction algorithms”. In: *ACM Computing Surveys* 39.2 (2007), page 4. http://dx.doi.org/10.1145/1242471.1242472.

[Rhe+10] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. “Prefix Tree Indexing for Similarity Search and Similarity Joins on Genomic Data”. In: *22nd International Conference on Scientific and Statistical Database Management (SSDBM’10)*. Volume 6187. LNCS. Heidelberg, Germany: Springer, 2010, pages 519–536. http://dx.doi.org/10.1007/978-3-642-13818-8_36.

[Ric+08] D. C. Richter, F. Ott, A. F. Auch, R. Schmid, and D. H. Huson. “MetaSim – A Sequencing Simulator for Genomics and Metagenomics”. In: *PLoS ONE* 3 (2008), e3373. http://dx.doi.org/10.1371/journal.pone.0003373.

[RO05] L. M. S. Russo and A. L. Oliveira. “Faster Generation of Super Condensed Neighbourhoods Using Finite Automata”. In: *12th International Conference on String Processing and Information Retrieval (SPIRE’05)*. Volume 3772. LNCS. Buenos Aires, Argentina: Springer, 2005, pages 246–255. http://dx.doi.org/10.1007/11575832_28.

[RO07] L. M. S. Russo and A. L. Oliveira. “Efficient generation of super condensed neighborhoods”. In: *Journal of Discrete Algorithms* 5.3 (2007), pages 501–513. http://dx.doi.org/10.1016/j.jda.2006.10.005.

[RO08] L. M. S. Russo and A. L. Oliveira. “A compressed self-index using a Ziv-Lempel dictionary”. In: *Information Retrieval* 11.4 (2008), pages 359–388. http://dx.doi.org/10.1007/s10791-008-9050-3.

[Rus+07] L. M. S. Russo, G. Navarro, and A. L. Oliveira. “Approximate String Matching with Lempel-Ziv Compressed Indexes”. In: *14th International Symposium on String Processing and Information Retrieval (SPIRE’07)*. Volume 4726. LNCS. Santiago, Chile: Springer, 2007, pages 264–275. http://dx.doi.org/10.1007/978-3-540-75530-2_24.

[Rus+08a] L. M. S. Russo, G. Navarro, and A. L. Oliveira. “Fully-Compressed Suffix Trees”. In: *8th Latin American Symposium on Theoretical Informatics (LATIN’08)*. Volume 4957. LNCS. Búzios, Brazil: Springer, 2008, pages 362–373. http://dx.doi.org/10.1007/978-3-540-78773-0_32.

[Rus+09a] L. M. S. Russo, G. Navarro, A. L. Oliveira, and P. Morales. “Approximate String Matching with Compressed Indexes”. In: *Algorithms* 2.3 (2009), pages 1105–1136. http://dx.doi.org/10.3390/a2031105.

[Rus+10] L. M. S. Russo, G. Navarro, and A. L. Oliveira. “Parallel and Distributed Compressed Indexes”. In: *21st Annual Symposium on Combinatorial Pattern Matching (CPM’10)*. Volume 6129. LNCS. New York City, NY, USA: Springer, 2010, pages 348–360. http://dx.doi.org/10.1007/978-3-642-13509-5_31.

[Rus+08b] L. M. Russo, G. Navarro, and A. L. Oliveira. “Dynamic Fully-Compressed Suffix Trees”. In: *19th Annual Symposium on Combinatorial Pattern Matching (CPM’08)*. Volume 5029. LNCS. Pisa, Italy: Springer, 2008, pages 191–203. http://dx.doi.org/10.1007/978-3-540-69068-9_19.

[Rus+09b] L. M. Russo, G. Navarro, and A. L. Oliveira. “Indexed Hierarchical Approximate String Matching”. In: *15th International Symposium on String Processing and Information Retrieval (SPIRE’08)*. Volume 5280. LNCS. Melbourne, Australia: Springer, 2009, pages 144–154. http://dx.doi.org/10.1007/978-3-540-89097-3_15.

[Sad00] K. Sadakane. “Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array”. In: *11th International Conference on Algorithms and Computation (ISAAC’00)*. Volume 1969. LNCS. Taipei, Taiwan: Springer, 2000, pages 410–421. http://dx.doi.org/10.1007/3-540-40996-3_35.

[Sad02] K. Sadakane. “Succinct representations of lcp information and improvements in the compressed suffix arrays”. In: *13th Annual Symposium on Discrete Algorithms (SODA’02)*. San Francisco, CA, USA: ACM/SIAM, 2002, pages 225–232. ISBN: 9780898715132. http://dl.acm.org/citation.cfm?id=545381.545410 (visited on 2014-11-06).

[Sad03] K. Sadakane. “New text indexing functionalities of the compressed suffix arrays”. In: *Journal of Algorithms* 48.2 (2003), pages 294–313. http://dx.doi.org/10.1016/S0196-6774(03)00087-7.

[Sad07] K. Sadakane. “Compressed Suffix Trees with Full Functionality”. In: *Theory of Computing Systems* 41.4 (2007), pages 589–607. http://dx.doi.org/10.1007/s00224-006-1198-x.

[SS01] K. Sadakane and T. Shibuya. “Indexing Huge Genome Sequences for Solving Various Problems”. In: *Genome Informatics Series* 12 (2001), pages 175–183. http://www.jsbi.org/pdfs/journal1/GIW01/GIW01F18.pdf (visited on 2014-11-06).

[Sam05] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., 2005. ISBN: 9780123694461.

[San09] P. Sanders. “Algorithm Engineering – An Attempt at a Definition”. In: *Efficient Algorithms*. Edited by S. Albers, H. Alt, and S. Näher. Volume 5760. LNCS. Springer, 2009, pages 321–340. http://dx.doi.org/10.1007/978-3-642-03456-5_22.

[SS03] K.-B. Schürmann and J. Stoye. *Suffix tree construction and storage with limited main memory*. Technical report. Universität Bielefeld, 2003. http://pub.uni-bielefeld.de/publication/1970475 (visited on 2014-11-06).

[SS07] K.-B. Schürmann and J. Stoye. “An incomplex algorithm for fast suffix array construction”. In: *Software – Practice and Experience* 37 (2007), pages 309–329. http://dx.doi.org/10.1002/spe.768.

[Sch04] T. Schürmann. “Bias analysis in entropy estimation”. In: *Journal of Physics A: Mathematical and General* 37.27 (2004), page L295. http://dx.doi.org/10.1088/0305-4470/37/27/L02.

[SG96] T. Schürmann and P. Grassberger. “Entropy estimation of symbol sequences”. In: *CHAOS – An Interdisciplinary Journal of Nonlinear Science* 6.3 (1996), pages 414–427. http://dx.doi.org/10.1063/1.166191.

[Sek+14] S. E. Seker, O. Altun, U. gur Ayan, and C. Mert. “A Novel String Distance Function Based on Most Frequent K Characters”. In: *International Journal of Machine Learning and Computing* 4.2 (2014), pages 177–182. http://dx.doi.org/10.7763/IJMLC.2014.V4.408.

[Sel80] P. H. Sellers. “The Theory and Computation of Evolutionary Distances: Pattern Recognition”. In: *Journal of Algorithms* 1.4 (1980), pages 359–373. http://dx.doi.org/10.1016/0196-6774(80)90016-4.

[Sew00] J. Seward. *bzip2 and libbzip2*. Website. 2000. http://www.bzip.org/ (visited on 2015-04-02).

[SM96] H. Shang and T. H. Merrett. “Tries for Approximate String Matching”. In: *IEEE Transactions on Knowledge and Data Engineering* 8.4 (1996), pages 540–547. http://dx.doi.org/10.1109/69.536247.

[Sha+48] C. E. Shannon, N. Petigara, and S. Seshasai. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27 (1948), pages 379–423. http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf (visited on 2014-11-06).

[Sha+04] H. Shatkay, J. Miller, C. Mobarry, M. Flanigan, S. Yooseph, and G. Sutton. “ThurGood: Evaluating Assembly-to-Assembly Mapping”. In: *Journal of Computational Biology* 11.5 (2004), pages 800–811. http://dx.doi.org/10.1089/cmb.2004.11.800.

[SJ08] J. Shendure and H. Ji. “Next-generation DNA sequencing”. In: *Nat. Biotechnol.* 26.10 (2008), pages 1135–1145. http://dx.doi.org/10.1038/nbt1486.

[Shr+14] A. M. S. Shrestha, M. C. Frith, and P. Horton. “A bioinformatician’s guide to the forefront of suffix array construction algorithms”. In: *Briefings in Bioinformatics* 15.2 (2014), pages 138–154. http://dx.doi.org/10.1093/bib/bbt081.

[Sin12] J. Singer. “A Wavelet Tree Based FM-Index for Biological Sequences in SeqAn”. Master thesis. Freie Universität Berlin, 2012. http://www.mi.fu-berlin.de/en/inf/groups/abi/theses/master_dipl/singer/ (visited on 2014-11-06).

[Sir+13a] E. Siragusa, D. Weese, and K. Reinert. “Fast and accurate read mapping with approximate seeds and multiple backtracking”. In: *Nucleic Acids Research* 41 (2013), e78. http://dx.doi.org/10.1093/nar/gkt005.

[Sir+13b] E. Siragusa, D. Weese, and K. Reinert. “Scalable String Similarity Search/Join with Approximate Seeds and Multiple Backtracking”. In: *Joint EDBT/ICDT 2013 Workshops (EDBT’13)*. Genoa, Italy: ACM, 2013, pages 370–374. http://dx.doi.org/10.1145/2457317.2457386.

[Sir09] J. Sirén. “Compressed Suffix Arrays for Massive Data”. In: *16th International Symposium on String Processing and Information Retrieval (SPIRE’09)*. Volume 5721. LNCS. Saariselkä, Finland: Springer, 2009, pages 63–74. http://dx.doi.org/10.1007/978-3-642-03784-9_7.

[Sir10] J. Sirén. “Sampled Longest Common Prefix Array”. In: *21st Annual Symposium on Combinatorial Pattern Matching (CPM’10)*. Volume 6129. LNCS. New York City, NY, USA: Springer, 2010, pages 227–237. http://dx.doi.org/10.1007/978-3-642-13509-5_21.

[Sir+09] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. “Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections”. In: *16th International Symposium on String Processing and Information Retrieval (SPIRE’09)*. Volume 5280. LNCS. Saariselkä, Finland: Springer, 2009, pages 164–175. http://dx.doi.org/10.1007/978-3-540-89097-3_17.

[Smi+87] K. D. Smith, K. E. Young, C. C. Talbot, and B. J. Schmeckpeper. “Repeated DNA of the human Y chromosome”. In: *Development* 101.Supplement (1987), pages 77–92. http://dev.biologists.org/content/101/Supplement/77.abstract (visited on 2015-04-02).

[SW81] T. F. Smith and M. S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pages 195–197. http://dx.doi.org/10.1016/0022-2836(81)90087-5.

[Sta11] T. Stadler. “Konstruktion von komprimierten Indizes für riesige Texte”. Master thesis. Technische Universität München, 2011. http://www14.in.tum.de/diplomarbeiten/abgeschlossen/2011-stadler.pdf.

[Ste+01] L. Stern, L. Allison, R. L. Coppel, and T. I. Dix. “Discovering patterns in plasmodium falciparum genomic DNA”. In: *Molecular and Biochemical Parasitology* 118.2 (2001), pages 175–186. http://dx.doi.org/10.1016/S0166-6851(01)00388-7.

[Sun90] D. M. Sunday. “A Very Fast Substring Search Algorithm”. In: *Communications of the ACM* 33.8 (1990), pages 132–142. http://dx.doi.org/10.1145/79173.79184.

[ST96] E. Sutinen and J. Tarhio. “Filtration with q-Samples in Approximate String Matching”. In: *7th Annual Symposium on Combinatorial Pattern Matching (CPM’96)*. Volume 1075. LNCS. Laguna Beach, California: Springer, 1996, pages 50–63. http://dx.doi.org/10.1007/3-540-61258-0_4.

[Tia+05] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel. “Practical methods for constructing suffix trees”. In: *The VLDB Journal* 14.3 (2005), pages 281–299. http://dx.doi.org/10.1007/s00778-005-0154-8.

[TS08] F. Transier and P. Sanders. “Compressed Inverted Indexes for In-Memory Search Engines”. In: *10th Workshop on Algorithm Engineering and Experiments (ALENEX’08)*. San Francisco, CA, USA: SIAM, 2008, pages 3–12. http://dx.doi.org/10.1137/1.9781611972887.1.

[TS10] F. Transier and P. Sanders. “Engineering basic algorithms of an in-memory text search engine”. In: *ACM Transactions on Information Systems* 29.1 (2010), 2:1–2:37. http://dx.doi.org/10.1145/1877766.1877768.

[Tsu10] D. Tsur. “Fast index for approximate string matching”. In: *Journal of Discrete Algorithms* 8.4 (2010), pages 339–345. http://dx.doi.org/10.1016/j.jda.2010.08.002.

[Ukk85] E. Ukkonen. “Finding approximate patterns in strings”. In: *Journal of Algorithms* 6.1 (1985), pages 132–137. http://dx.doi.org/10.1016/0196-6774(85)90023-9.

[Ukk92] E. Ukkonen. “Approximate string matching with q-grams and maximal matches”. In: *Theoretical Computer Science* 92.1 (1992), pages 191–211. http://dx.doi.org/10.1016/0304-3975(92)90143-4.

[Ukk93] E. Ukkonen. “Approximate string-matching over suffix trees”. In: *4th Annual Symposium on Combinatorial Pattern Matching (CPM’93)*. Volume 684. LNCS. Padova, Italy: Springer, 1993, pages 228–242. http://dx.doi.org/10.1007/BFb0029808.

[Ukk95] E. Ukkonen. “On-Line Construction of Suffix Trees”. In: *Algorithmica* 14.3 (1995), pages 249–260. http://dx.doi.org/10.1007/BF01206331.

[Umb+11] C. Umbel, R. Ellis, and R. Mull. *natural for Node.js*. Website. 2011. https://github.com/NaturalNode/natural (visited on 2015-04-03).

[VL09] R. Vernica and C. Li. "Efficient Top-k Algorithms for Fuzzy Search in String Collections". In: *1st International Workshop on Keyword Search on Structured Data (KEYS'09)*. Providence, Rhode Island: ACM, 2009, pages 9–14. http://dx.doi.org/10.1145/1557670.1557677.

[Väl+09] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. "Engineering a compressed suffix tree implementation". In: *Journal of Experimental Algorithmics* 14 (2009), pages 4.2–4.23. http://dx.doi.org/10.1145/1498698.1594228.

[Wan+14] S. Wandelt et al. "State-of-the-art in String Similarity Search and Join". In: *SIGMOD Records* 43.1 (2014), pages 64–76. http://dx.doi.org/10.1145/2627692.2627706.

[Wat+13] Y. Watanuki, K. Tamura, H. Kitakami, and Y. Takahashi. "Parallel processing of approximate sequence matching using disk-based suffix tree on multi-core CPU". In: *6h International Workshop on Computational Intelligence Applications (IWCIA'13)*. IEEE Computer Society, 2013, pages 137–142. http://dx.doi.org/10.1109/IWCIA.2013.6624801.

[Wee06] D. Weese. "Entwurf und Implementierung eines generischen Substring-Index". Diploma thesis. Humboldt-Universität, 2006. http://publications.mi.fu-berlin.de/457/ (visited on 2014-11-06).

[Wee12] D. Weese. "Indices and Applications in High-Throughput Sequencing". PhD thesis. Freie Universität Berlin, 2012. http://publications.mi.fu-berlin.de/1288/ (visited on 2014-11-06).

[Wei73] P. Weiner. "Linear Pattern Matching Algorithms". In: *14th Annual Symposium on Switching and Automata Theory (SWAT'73)*. Iowa City, IA, USA: IEEE Computer Society, 1973, pages 1–11. http://dx.doi.org/10.1109/SWAT.1973.13.

[Wik15] Wikibooks. *More C++ Idioms / Clear-and-minimize*. Website. 2015. http://en.wikibooks.org/wiki/More%20C%2B%2B%20Idioms/Clear-and-minimize?oldid=2582039 (visited on 2015-04-02).

[Wil03] K. Willets. "Full-Text Searching & the Burrows-Wheeler Transform". In: *Dr. Dobb's Journal* 28 (2003), pages 48–53. http://www.drdobbs.com/architecture-and-design/full-text-searching-the-burrows-wheeler/184405504 (visited on 2014-11-06).

[Win90] W. E. Winkler. "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage". In: *Survey Research Methods Section (ASA'90)*. U.S. Bureau of the Census. American Statistical Association, 1990, pages 354–359. http://www.amstat.org/sections/SRMS/Proceedings/papers/1990_056.pdf (visited on 2014-11-06).

[Win06] W. E. Winkler. *Overview of Record Linkage and Current Research Directions*. Research Report Series RRS2006/02. U.S. Census Bureau, Statistical Research Division, 2006. https://www.census.gov/srd/papers/pdf/rrs2006-02.pdf (visited on 2014-11-06).

[Won+06] J.-I. Won, S. Park, J.-H. Yoon, and S.-W. Kim. "An efficient approach for sequence matching in large DNA databases". In: *Journal of Information Science* 32.1 (2006), pages 88–104. http://dx.doi.org/10.1177/0165551506059229.

[Won+07] S.-S. Wong, W.-K. Sung, and L. Wong. “CPS-tree: A Compact Partitioned Suffix Tree for Disk-based Indexing on Large Genome Sequences”. In: *23rd International Conference on Data Engineering (ICDE’07)*. Istanbul, Turkey: IEEE Computer Society, 2007, pages 1350–1354. http://dx.doi.org/10.1109/ICDE.2007.369009.

[Woo+11] F. Wood, J. Gasthaus, C. Archambeau, L. James, and Y. W. Teh. “The Sequence Memoizer”. In: *Communications of the ACM* 54 (2 2011), pages 91–98. http://dx.doi.org/10.1145/1897816.1897842.

[WM92a] S. Wu and U. Manber. “Agrep – A Fast Approximate Pattern-Matching Tool”. In: *USENIX Winter 1992 Technical Conference*. San Francisco, CA, USA: USENIX Association, 1992, pages 153–162. https://www.usenix.org/legacy/publications/library/proceedings/wu.pdf (visited on 2014-11-06).

[WM92b] S. Wu and U. Manber. “Fast text searching allowing errors”. In: *Communications of the ACM* 35.10 (1992), pages 83–91. http://dx.doi.org/10.1145/135239.135244.

[WM94] S. Wu and U. Manber. *A Fast Algorithm for Multi-pattern Searching*. Technical report TR94-17. Department of Computer Science, University of Arizona, 1994. ftp://ftp.cs.arizona.edu/reports/1994/TR94-17.ps.

[YS11] K. Yang and J. C. Setubal. “A whole genome simulator of prokaryote genome evolution”. In: *ACM International Conference on Bioinformatics, Computational Biology and Biomedicine (BCB’11)*. Chicago, IL, USA, 2011, pages 508–510. http://dx.doi.org/10.1145/2147805.2147885.

[Yan+08] X. Yang, B. Wang, and C. Li. “Cost-based variable-length-gram selection for string collections to support approximate queries efficiently”. In: *International Conference on Management of Data (SIGMOD’08)*. Vancouver, Canada: ACM, 2008, pages 353–364. http://dx.doi.org/10.1145/1376616.1376655.

[Yan+10] Z. Yang, J. Yu, and M. Kitsuregawa. “Fast Algorithms for Top-k Approximate String Matching”. In: *24th AAAI Conference on Artificial Intelligence (AAAI’10)*. Atlanta, GA, USA: AAAI Press, 2010, pages 1467–1473. http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1939 (visited on 2014-11-06).

[Yos12] N. Yoshinaga. *cedar – C++ implementation of efficiently-updatable double-array trie*. Website, Kitsuregawa, Toyoda Lab., IIS, University of Tokyo. 2012. http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/ (visited on 2014-11-06).

[Zha+10] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. “Bed-tree: An All-purpose Index Structure for String Similarity Search Based on Edit Distance”. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD’10)*. Indianapolis, Indiana, USA: ACM, 2010, pages 915–926. http://dx.doi.org/10.1145/1807167.1807266.

[ZL78] J. Ziv and A. Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Transactions on Information Theory* 24.5 (1978), pages 530–536. http://dx.doi.org/10.1109/TIT.1978.1055934.

[ZD95] J. Zobel and P. W. Dart. “Finding Approximate Matches in Large Lexicons”. In: *Software – Practice and Experience* 25.3 (1995), pages 331–345. http://dx.doi.org/10.1002/spe.4380250307.

[ZM06] J. Zobel and A. Moffat. “Inverted files for text search engines”. In: *ACM Computing Surveys* 38.2 (2006), pages 1–56. http://dx.doi.org/10.1145/1132956.1132959.