TUM

INSTITUT FÜR INFORMATIK

Text Indexing with Errors

Moritz G. Maaß and Johannes Nowak

TUM-I0503
März 05

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-03-I0503-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Text Indexing with Errors

Moritz G. Maaß[*] Johannes Nowak

{maass,nowakj}@informatik.tu-muenchen.de

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany

Abstract

In this paper we address the problem of constructing an index for a text document or a collection of documents to answer various questions about the occurrences of a pattern when allowing a constant number of errors. In particular, our index can be built to report all occurrences, all positions, or all documents where a pattern occurs in time linear in the size of the query string and the number of results. This improves over previous work where the look-up time was either not linear or depended upon the size of the document corpus. Our data structure has size $O(n \log^d n)$ on average and with high probability for input size n and queries with up to d errors. Additionally, we present a trade-off between query time and index complexity that achieves worst-case bounded index size and preprocessing time with linear look-up time on average.

1 Introduction

A text index is a data structure prepared for a document or a collection of documents that facilitates efficient queries for the occurrences of a pattern. Text indexing is becoming increasingly important. The amount of textual data available, e.g., in the Internet or in biological databases, is tremendous and growing. The sheer size of the textual data makes the use of indexes for efficient on-line queries vital. On the other hand, the nature of the data frequently calls for error-tolerant methods (called approximate pattern matching): data on the Internet is often less carefully revised and contains more typos than text published in classical media with professional editorial staff; biological data is often erroneous due to mistakes in its experimental generation. Moreover, in a biological context, error-tolerant matching is useful even for immaculate data, e.g., for similarity searches. For on-line searches, where no preprocessing of the document corpus is done, there is a variety of algorithms available for many different error models (see, e.g., the survey [Nav01]). Recently, some progress has been made towards the construction of error-tolerant text indexes [AKL$^+$00, BGW00, CGL04], but in general the problem remains open. In particular, currently no method with optimal look-up time—linear in the pattern length and the number of outputs—is known. We fill this gap with our new indexing method.

When indexing a document (or a collection C of documents) of total size n and performing a query for a pattern of length m allowing d errors, the relevant parameters are the index size, the index

[*] Research supported in part by DFG, grant Ma 870/5-1 (Leibnizpreis Ernst W. Mayr).

construction time, the look-up time, and the error model. Usually, the least important of these parameters is the preprocessing time. Depending on the application, either size or query time dominates. We consider output-sensitive algorithms here, i.e., the complexity of the algorithms is allowed to depend on an additional parameter occ counting the number of outputs, e.g., occurrences of a pattern. The natural lower bound, linear $\Theta(n)$ size (preprocessing time) and linear $\Theta(m + \mathrm{occ})$ look-up time, can be reached[1] for exact matching (e.g., [Wei73, McC76, Ukk95, Far97]). For edit or Hamming distance, no index with look-up time $O(m + \mathrm{occ})$ and size $O(n \log^l n)$ for $l = O(1)$ even for a small number of errors is known. In all reasonable-size indexes the look-up time depends on n or is not linear in m.

1.1 Our Results

We present and analyze a new index structure for approximate pattern matching problems allowing a constant number of $d = O(1)$ errors. The method works for various problem flavors, e.g., approximate text indexing (full text indexing), approximate dictionary look-up indexing (word based indexing), and approximate document collection indexing (see Section 2.4). For all of these problems we achieve an optimal worst-case look-up time of $O(m + \mathrm{occ})$ employing an index that uses $O(n \log^d n)$ additional space and requires preprocessing time $O(n \log^{d+1} n)$, both on average and with high probability. For approximate dictionary look-up these bounds even improve to $O(|C| \log^d |C|)$ additional space and $O(|C| \log^{d+1} |C|)$ preprocessing time, where $|C|$ is the number of documents in the collection. Our data structure is based on a compact trie representation of the text corpus. This can either be a compact trie, a suffix tree, or a generalized suffix tree. From this tree further error trees are constructed (see Section 3.2). For the efficient retrieval of results, range queries are prepared on the leaves of the trees (see Section 2.5). The average case analysis is based on properties of mixing ergodic stationary sources which encompass a wide range of probabilistic models such as stationary ergodic Markov sources (see, e.g., [Szp00]). To our knowledge, this yields the first reasonable sized indexes achieving optimal look-up time. Additionally, we present a trade-off between query time and index complexity, achieving worst-case bounded index size $O(n \log^d n)$ and preprocessing time $O(n \log^{d+1} n)$ while having linear look-up time $O(m + \mathrm{occ})$ on average, i.e., we can have a worst-case bound either on the size or on the query time and an average-case bound on the other.

1.2 Related Work

A survey on text indexing is given by Navarro et al. [NBYST01]. For the related nearest neighbor problem see the survey by Indyk [Ind04]. Previous results for text indexing (on a single text of length n) are summarized in the table below. Navarro and Baeza-Yates [NBY00] present a method with $O(n^\epsilon)$, $\epsilon < 1$, average look-up time for general edit distance. A similar result is reported by Myers [Mye94], who describes an algorithm with expected look-up time $O(n^\epsilon \log n)$ for d errors. In a certain range, this is also sublinear. Both algorithms require $O(n)$ space. Another approach is taken by Chávez and Navarro [CN02]. Using a metric index they achieve a look-up time $O(m \log^2 n + m^2 + \mathrm{occ})$ with an index of size $O(n \log n)$ with $O(n \log^2 n)$ construction time, where all complexities are achieved on average. A backtracking approach on suffix trees was proposed by Ukkonen [Ukk93] having look-up time $O(mq \log q + \mathrm{occ})$ and space requirement $O(mq)$ with $q = \min\{n, m^{d+1}\}$. This was improved by Cobbs [Cob95] to look-up time $O(mq + \mathrm{occ})$ and space

[1] We assume a uniform cost model throughout this work.

$O(q+n)$. Amir et al. [AKL$^+$00] describe an index for $d = 1$ and edit distance. This was later improved by Buchsbaum et al. [BGW00] to $O(n \log n)$ index size and $O(m \log\log n + \mathrm{occ})$ query time. For Hamming distance, an index with $O(m + \mathrm{occ})$ look-up time using $O(n \log n)$ space on average can be constructed [Now04]. This was generalized to edit distance in [MN05]. Recently, Cole et al. [CGL04] proposed a structure allowing a constant number d of errors, which works for don't cares (wild cards) and edit distance (the table gives the result for edit distance). For Hamming distance the dictionary look-up problem can be solved with a compact trie with $O(|C|)$ extra space and $O(\log^{d+1}|C|)$ average look-up time [Maa04]. A somewhat different approach is taken by Gabriele et al. [GMRS03], for a restricted Hamming distance allowing d mismatches in every window of r characters, they describe an index that has average size $O(n \log^l n)$, for some constant l and average look-up time $O(m + \mathrm{occ})$.

Errors	Model	Query Time	Index Size	Prep. Time	Literature
$d=0$	exact	$O(m+\mathrm{occ})$	$O(n)$	$O(n)$	Weiner 1973
$d=1$	edit	$O(m \log n \log\log n+\mathrm{occ})$	$O(n \log^2 n)$	$O(n \log^2 n)$	Amir et al. 2000
$d=1$	edit	$O(m \log\log n+\mathrm{occ})$	$O(n \log n)$	$O(n \log n)$	Buchsbaum et al. 2000
$d=1$	edit	$O(m+\mathrm{occ})$	$O(n \log n)$ (avg,whp)	$O(n \log^2 n)$ (avg,whp)	MN 2004
$d=O(1)$	edit	$O(m+\log^d n \log\log n+\mathrm{occ})$	$O(n \log^d n)$	$O(n \log^d n)$	Cole et al. 2004
$d=O(1)$	edit	$O(n^\epsilon)$	$O(n)$	$O(n)$	Navarro et al. 2000
$d=O(1)$	edit	$O(kn^\epsilon \log n)$	$O(n)$	$O(n)$	Myers 1994
$d=O(1)$	edit	$O(m \log^2 n+m^2+\mathrm{occ})$ (avg)	$O(n \log n)$ (avg)	$O(n \log^2 n)$ (avg)	Chávez et al. 2002
$d=O(1)$	edit	$O(m \min\{n, m^{d+1}\}+\mathrm{occ})$	$O(\min\{n, m^{d+1}\}+n)$	$O(\min\{n, m^{d+1}\}+n)$	Cobbs 1995 (Ukkonen 1993)
$d=O(1)$	Ham.	$O(\log^{d+1} n)$, (avg)	$O(n)$	$O(n)$	M 2004
$d=O(1)$	edit	$O(m+\mathrm{occ})$	$O(n \log^d n)$ (avg,whp)	$O(n \log^{d+1} n)$ (avg,whp)	This paper
$d=O(1)$	edit	$O(m+\mathrm{occ})$, (avg,whp)	$O(n \log^d n)$	$O(n \log^{d+1} n)$	
d mismatches in a window of length r		$O(m+\mathrm{occ})$, (avg)	$O(n \log^l n)$, (avg)	$O(n \log^l n)$, (avg)	Gabriele et. al 2003

The approach of Cole et al. [CGL04] can also be used for the approximate dictionary indexing problem with similar complexities. Previous results on dictionary indexing only allowed one error. For approximate dictionary indexing with a set of n strings of length m, Yao and Yao [YY97] (earlier version in [YY95]) present and analyze a data structure that achieves a query time of $O(m \log\log n + \mathrm{occ})$ and space $O(N \log m)$. Also for Hamming distance and dictionaries of n strings of length m each, Brodal and Gąsieniec [BG96] present an algorithm based on tries that uses similar ideas than those in [AKL$^+$00]. Their data structure has size and preprocessing time $O(N)$ and supports queries with one mismatch in time $O(m)$. Brodal and Venkatesh [BV00] analyze the problem in a cell-probe model with word size m. The query time of their approach is $O(m)$ using $O(N \log m)$ space. The data structure can be constructed in randomized expected time $O(Nm)$.

The exact version of the indexing problem are much better understood. There are myriads of different indexing methods besides the already mentioned suffix trees. Among these, the suffix array [MM93] is the most prominent. It is smaller in practice, can be built in linear time [KSPP03, KA03, KS03], and allows linear time look-ups [AOK02]. Further research is aimed at indexes using only linear space in a bit model [FM00, GV00]. Regarding the index space, a linear lower bound has also been proven for the exact indexing problem [DLO01].

For the approximate (i.e., error-tolerant) indexing problem the results is much scarcer. For the case of a constant number of errors, our work together with [CGL04] and [Maa04] seems to indicate that—compared to exact searching—an additional complexity factor of $O(\log^d n)$ is inherent. However, lower bounds for the approximate indexing problem do not seem easy to achieve. Using asymmetric communication complexity some bounds for nearest neighbor search in the Hamming cube can be shown [BOR99, BR00, BV02], but these do not apply to the case where a linear number (in the size of the pattern) of probes to the index is allowed. The lower bound in [BV02] is derived from ordered binary decision diagrams (OBDDs) and assumes that a pattern is only ever read in one direction. The information theoretic method of [DLO01] also seems to fall short because approximate look-up does not improve compression and there is no restriction on the look-up time.

2 Preliminaries

2.1 Strings and String Distances

Let Σ be any finite alphabet and let $|\Sigma|$ denote its size. We consider $|\Sigma|$ to be constant. Σ^* is the set of all strings including the empty string ε. Let $t = t[1] \cdots t[n] \in \Sigma^n$ be a string of length $|t| = n$. We denote by uv (and sometimes $u{\cdot}v$) the concatenation of the strings u and v. If $t = uvw$ with $u, v, w \in \Sigma^*$ then u is a prefix, v a substring, and w a suffix of t. We define $t[i..j] = t[i]t[i{+}1] \cdots t[j]$, $\mathrm{pref}_k(t) = t[1..k]$, and $\mathrm{suff}_k(t) = t[k..n]$ (with $t[i..j] = \varepsilon$ for $i > j$). For $u, v \in \Sigma^*$ we let $u \sqsubseteq_{\mathsf{pref}} v$ denote that $u = \mathrm{pref}_{|u|}(v)$. For $S \subseteq \Sigma^*$ and $u \in \Sigma^*$ we let $u \in_{\mathsf{pref}} S$ denote that there is $v \in S$ such that $u \sqsubseteq_{\mathsf{pref}} v$. Let $u \in \Sigma^*$ be the longest common prefix of two strings $v, w \in S$. We define $\mathrm{maxpref}(S) = |u|$. The size of S is defined as $\sum_{u \in S} |u|$.

We use the well-known *edit distance* (Levenshtein distance [Lev65]) to measure distances between strings. The edit distance of two strings u and v, $\mathrm{d}(u, v)$, is defined as the minimum number of *edit operations* (*deletions*, *insertions*, and *substitutions*) that transform u into v. We restrict our attention to the unweighted model, i.e., every operation is assigned a cost of one. The edit distance between two strings $u, v \in \Sigma^*$ is easily computed using dynamic programming in $O(|u| \cdot |v|)$ using the following recurrence:

$$\mathrm{d}\left(u[1..k], v[1..l]\right) = \min \left\{ \begin{array}{l} \mathrm{d}\left(u[1..k], v[1..l-1]\right) + 1 \\ \mathrm{d}\left(u[1..k-1], v[1..l]\right) + 1 \\ \mathrm{d}\left(u[1..k-1], v[1..l-1]\right) + \hat{\delta}_{u[k],v[l]} \end{array} \right\} , \qquad (1)$$

where $\hat{\delta}_{u[k],v[l]} = 0$ if and only if $u[k] = v[l]$ and $\hat{\delta}_{u[k],v[l]} = 1$ otherwise. *Hamming distance* [Ham50] can be seen as a simplified version of edit distance. For two strings $u, v \in \Sigma^*$, the Hamming distance is either infinite if the strings have different lengths, or it is defined as

$$\mathrm{d}_{\mathsf{Ham}}(u, v) = \sum_{l=1}^{|u|} \hat{\delta}_{u[l],v[l]} \quad . \qquad (2)$$

We restrict our attention to edit distance, using Hamming distance instead would even simplify the matter.

2.2 Tries

For fast look-up, strings can be stored in a *trie*. A trie $\mathcal{T}$ for a set of strings $S \subset \Sigma^*$ is a rooted tree with edges labeled by characters from Σ. All outgoing edges of a node are labeled by different characters (*unique branching criterion*). Each path from the root to a leaf can be read as a string from S. Let u be the string constructed from concatenating the edge labels from the root to a node x. We define $\mathrm{path}(x) = u$. The string depth of node x is defined by $\mathrm{depth}(x) = |\mathrm{path}(x)|$. The *word set* of $\mathcal{T}$ denoted $\mathrm{words}(\mathcal{T})$ is the set of all strings u, such that $u \sqsubseteq_{\mathsf{pref}} \mathrm{path}(x)$ for some $x \in \mathcal{T}$. For a node in $x \in \mathcal{T}$ we define $\mathcal{T}_x$ to be the sub-trie rooted at x. Defining $\mathrm{tails}_u(S) = \{v \mid uv \in S\}$, we can characterize the sub-trie rooted at node x with $u = \mathrm{path}(x)$ by $T_x = \mathcal{T}(\mathrm{tails}_u(S))$. For convenience we also denote $T_u = T_{\mathrm{path}(x)} = T_x$. The string height of $\mathcal{T}$ is defined as $\mathrm{height}(\mathcal{T}) = \max\{\mathrm{depth}(x) \mid x \text{ is an inner node of } \mathcal{T}\}$; it is the same as $\mathrm{maxpref}(\mathrm{words}(\mathcal{T}))$.

A *compact trie* is a trie where nodes with only one outgoing edge have been eliminated and edges are labeled by strings from Σ^+ (more precisely, they are labeled by pointers into the underlying strings). Eliminated nodes can be represented by *virtual nodes*, i.e., a base node, the character of the outgoing edge, and the length. A similar representation is used in [Ukk95], called reference pairs (i.e., if x is a trie-node and y is the node in the compact trie representing the maximal length prefix $\mathrm{path}(y) \sqsubseteq_{\mathsf{pref}} \mathrm{path}(x)$, then x can be represented by $(y, u[|\mathrm{path}(y)|+1], |\mathrm{path}(x)| - |\mathrm{path}(y)|)$). All previous definitions for tries apply to compact tries as well, possibly using virtual instead of existing nodes. We only use compact tries, hence, we denote the compact trie for the string set S by $\mathcal{T}(S)$.

2.3 Weak Tries

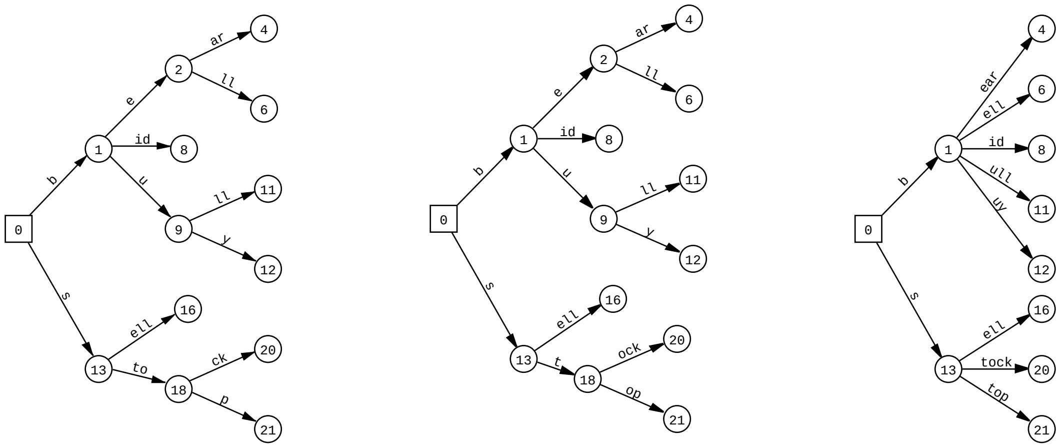
For ease of representation, we consider the strings in underlying sets of any trie like data structure to be independent and make no use of any intrinsic relations between them. A suffix tree for the string t can also be regarded as a trie for the set $S = \{u|u \text{ is a suffix of } t\}$ which can be represented in size $O(|t|)$. The major advantage of the suffix tree is that (by using the properties of S) it can be built in time $O(|t|)$, while building a trie for the suffixes of t may take time $O(l|t|)$ where l is the length of the longest common repeated substring in t. For our index data structure, this additional cost will only be marginal. In the following let $\mathcal{T}(S)$ denote the compact trie for the string set $S \subset \Sigma^+$.

To reduce the size of our index in the worst-case, we later restrict the search to patterns with a maximal length l. We then only need to search the tries up to the depth l. The structure from this threshold to the leaves is not important. This concept is captured in the following definition of weak tries.

Definition 2.1 (Weak Trie). *For $l > 0$, the l-weak trie for a set of strings $S \subset \Sigma^*$ is a rooted tree with edges labeled by characters from Σ. For any node with a depth less than l, all outgoing edges are labeled by different characters, and there are no branching nodes with a depth of more than l. Each path from the root to a leaf can be read as a string from S.*

Up to level l, all previous definitions for (compact) tries carry over to (compact) weak tries. The remaining branches (in comparison to a compact trie) are all at level l. By $\mathcal{W}_l(S)$ we denote a compact l-weak trie for the set of strings S. Note that $\mathcal{W}_{\mathrm{maxpref}(S)}(S) = \mathcal{T}(S)$. In any case, the largest depth of a branching node in an l-weak trie is l.

Figure 1 shows examples of weak tries. The height of the trie in Figure 1(a) is three, thus the 3-weak trie is the same as the compact trie. Since $\mathrm{maxpref}(S)$ is the height of the trie for the string set S, the weak trie $\mathcal{W}_{\mathrm{maxpref}(S)}(S)$ is just the compact trie $\mathcal{T}(S)$.



(a) Compact trie (b) 2-Weak trie (c) 1-Weak trie

Figure 1: Examples of a compact trie, a 1-weak trie, and a 2-weak trie for the prefix-free set of strings `bear`, `bell`, `bid`, `bull`, `buy`, `sell`, `stock`, `stop`.

The l-weak trie for a set S can easily be implemented in size $O(|S|)$ by representing edge labels with pointers into the strings of S: Each string in S generates a leaf and so there are at most $O(|S|)$ leaves, $O(|S|)$ inner nodes, and $O(|S|)$ edges.

Definition 2.1 guarantees that weak tries are unique with respect to a string set S (except for the order of the children): Since the trie is compact, by the unique branching criterion, the nodes up to depth l are uniquely defined through the prefixes of length l of the strings in S. Each element in S has a prefix of length l that uniquely corresponds to a path of length l. If there is more than one string with a certain prefix, u, then these are all represented by leaves under a node p with $\mathrm{path}(p) = u$.

Up to level l, all previous definitions for (compact) tries carry over to (compact) weak tries. Weak tries are not necessarily unique with respect to a string set S. By $\mathcal{W}_l(S)$ we denote an (arbitrary) compact l-weak trie. Note that $\mathcal{W}_{\mathrm{maxpref}(S)}(S) = \mathcal{T}(S)$ (since $\mathrm{maxpref}(S)$ is the string height of $\mathcal{T}(S)$).

2.4 Approximate Indexing Problems

Pattern matching problems come in various flavors. We focus on the case where a single pattern P is to be found in a database consisting of a text T or a collection of texts C. The database is considered static and can be preprocessed to allow fast dynamic, on-line queries. There appear various definitions of approximate text indexing in the literature. The broader definition just requires the index to "speed up searches" [NBYST01], while a stricter approach requires to answer on-line queries "in time proportional to the pattern length and the number of occurrences" [AKL$^+$00]. We follow the latter approach.

Problem 2.2 (d-Approximate Text Indexing (d-ATI)). *Given a text $T \in \Sigma^*$, preprocess T such that upon a query for a pattern $P \in \Sigma^*$ we can retrieve*

*a. all occurrences (i, j) such that a substring $T[i..j]$ matches P with at most d errors (*reporting occurrences*),* *b. all positions i such that a substring $T[i\,..j]$ matches P with at most d errors (*reporting positions*).*

Problem 2.3 (d-Approximate Dictionary Indexing (d-ADI)). *Given a finite collection of strings $C \subset \Sigma^*$, preprocess C such that upon a query for a pattern P we can retrieve*

*a. all occurrences $(s, 1, i)$, $s \in C$, such that a prefix $\mathrm{pref}_i(s)$ matches P with at most d errors (*reporting occurrences*),*

*b. all strings $s \in C$ such that a prefix $t \sqsubseteq_{\mathsf{pref}} s$ matches P with at most d errors (*reporting positions*),*

*c. all strings $s \in C$ that match P with at most d errors (*reporting full hits*).*

Problem 2.4 (d-Approximate Document Collection Indexing (d-ADCI)). *Given a finite collection of strings $C \subset \Sigma^*$, preprocess C such that upon a query for a pattern P we can retrieve*

*a. all occurrences (s, i, j), $s \in C$, such that $s[i\,..j]$ matches P with at most d errors (*reporting occurrences*),*

*b. all positions (s, i), $s \in C$, such that a substring $s[i\,..j]$ matches P with at most d errors (*reporting positions*),*

*c. all strings s such that a substring $s[i\,..j]$ matches P with at most d errors (*reporting documents*).*

To ease further exposition, we take a unified view on the three indexing categories by considering our text database to be a set S of n strings, called the *base set*. We let S be either one of the following: (i) the set of all suffixes of T for d-ATI, (ii) the collection of strings C for d-ADI, (iii) the set of all suffixes of all strings in C for d-ADCI. Observe that in all instances in addition to the size of the (originally) given strings a trie for the underlying collection consumes $O(n)$ space by using pointers into strings instead of substrings.

2.5 Range Queries

A basic tool needed for our algorithm are *range queries*. The following range queries are used to efficiently select the correct occurrences of a pattern depending on the problem type.

Problem 2.5 (Bounded Value Range Query (BVRQ)). *An array A of size n (indexed 1 through n) containing integer values is to be preprocessed for answering bounded value range queries efficiently. A BVRQ (i, j, b) asks to find all indices $L \subseteq [i, j]$ such that the value in A is less than or equal to b, i.e., $L = \{l \mid i \leq l \leq j$ and $A[l] \leq b\}$.*

Problem 2.6 (Colored Range Query (CRQ)). *An array A of size n (indexed 1 through n) containing integer values is to be preprocessed for answering colored range queries efficiently. A CRQ (i, j) asks to find the distinct set of different numbers that appear in the interval $[i, j]$ of A, i.e., to return the set $C = \{A[c] \mid i \leq c \leq j\}$.*

The BVRQ problem can be solved with $O(n)$ preprocessing time and space and $O(|L|)$ query time [Mut02], based on the well-known range minimum query (RMQ) problem which can be solved with $O(n)$ preprocessing and $O(1)$ query time [GBT84]. The CRQ problem can also be solved with $O(n)$ preprocessing time and space and $O(|C|)$ query time [Mut02].

2.6 A Closer Look at the Edit Distance

The presented algorithms work for edit distance with unit cost up to a constant number of errors d. Any submodel of edit distance can be used as well, but we only describe the edit distance version. To understand the basic ideas first, it might be helpful to read this chapter replacing edit distance by Hamming distance.

Our indexing structure is based on the idea of computing certain strings which are within a specified distance to elements from our base set S or sets derived from S where the errors occur in prefixes of a bounded length. Therefore, we have to establish close ties between the length of minimal prefixes and the number of mismatches therein.

The following definition captures the minimal prefix length of a string u that contains all errors with respect to a string v.

Definition 2.7 (k-Minimal Prefix Length). *For two strings $u, v \in \Sigma^*$ with $\mathrm{d}(u, v) = k$ we define*

$$\begin{aligned}&\mathrm{minpref}_{k,u}(v)\\&\quad= \min\left\{l \mid \mathrm{d}\left(\mathrm{pref}_l(u), \mathrm{pref}_{l+|v|-|u|}(v)\right) = k \textit{ and } \mathrm{suff}_{l+1}(u) = \mathrm{suff}_{l+|v|-|u|+1}(v)\right\} \quad . \qquad (3)\end{aligned}$$

Note that, for Hamming distance, $\mathrm{minpref}_{k,u}(v)$ is the position of the last mismatch.

For a more detailed understanding of edit distance, it is helpful to consider the *edit graph*. The edit graph for the transformation of the string u into the string v contains a vertex[2] for each pair of prefixes. An arc connects two vertices if the prefixes represented by the vertices differ by at most one character. Each arc is labeled by a weight corresponding to equation (1), i.e., the weight is zero if and only if the prefixes of the source vertex are both extended by the same character to form the prefixes of the target vertex (a match). Otherwise, the weight of the arc is one, corresponding to a substitution (diagonal arcs), a deletion (horizontal arcs), or an insertion (vertical arcs). The vertex representing the empty prefixes is the start vertex and it is labeled with the weight zero. Each vertex in the edit graph is labeled with the weight of a lightest (or shortest if we consider weights as distances) path connecting it to the start vertex. The vertex representing both strings is the end vertex. Its label is the edit distance between both strings. We call an arc relevant if it lies on a shortest path from the start vertex to another vertex. The *relevant edit graph* contains only the relevant arcs. We denote the relevant edit graph for transforming u into v by $G^{\mathsf{rel}}_{u,v}$. An *edit path* is any path in the relevant edit graph connecting the start with the end vertex. Each path connecting the start with the end vertex corresponds to a minimal set of edit operations to transform one string into the other.

Recall that the edit distance is defined as the minimal number of operations necessary to transform a string u into another string v. It is convenient to define the operators del, ins, and sub of type $\Sigma \cup \{\varepsilon\} \to \Sigma \cup \{\varepsilon\}$. If, for two strings $u, v \in \Sigma^*$, we have distance $\mathrm{d}(u, v) = k$, then there exist one or more sequences of operations $(op_1, op_2, \ldots, op_k)$, $op_i \in \{\mathrm{del}, \mathrm{ins}, \mathrm{sub}\}$, such that $v = op_k(op_{k-1}(\cdots op_1(u)\cdots))$. We call $u(i) = op_i(\cdots op_1(u)\cdots)$ the i-th *edit stage*. Each operation op_i in the sequence changes the current string at a position $\mathrm{pos}\,(op_i)$. An insertion changing $u = u_1\cdots u_{i-1}u_iu_{i+1}\cdots u_m$ to $u_1\cdots u_{i-1}au_i\cdots u_m$ has position i, a deletion changing u to $u_1\cdots u_{i-1}u_{i+1}\cdots u_m$ has position i, and a substitution changing u to $u_1\cdots u_{i-1}au_{i+1}\cdots u_m$ also has position i. We call a sequence $\rho(u, v) = (op_1, op_2, \ldots, op_k)$ of edit operations an *ordered edit sequence* if the operations are applied from left to right, i.e., for op_i and op_{i+1} we have $\mathrm{pos}\,(op_i) \le \mathrm{pos}\,(op_{i+1})$

[2] To avoid confusion, we call elements of the edit graph vertices and arcs and elements of the trees of our index data structure nodes and edges.

if op_i is a deletion, and $\mathrm{pos}\,(op_i) < \mathrm{pos}\,(op_{i+1})$ otherwise. Observe that changing the order or the operations also effects the positions in the string where the operations apply. For a fixed set of operations there is a unique order (except for swapping identical del_i-operations). The charm of ordered edit sequences lies in the fact that they transform one string into the other in a well-defined way.

Note that there may be exponentially (in the number of errors) many edit paths: Consider the strings $\mathtt{ba}^m\mathtt{b}$ and $\mathtt{ba}^n\mathtt{b}$ which have distance $d = n - m$ for $n > m$. There are $\binom{n-2}{d}$ possibilities to remove the additional $\mathtt{a}$s and, thus, equally many edit paths.

Lemma 2.8 (One-to-One Mapping between Paths and Edit Sequences). *Let $u, v \in \Sigma^*$ be such that $\mathrm{d}(u, v) = k$. Each ordered edit sequence $\rho(u, v) = (op_1, \ldots, op_k)$ corresponds uniquely to an edit path $\pi = (p_1, \ldots, p_m)$ in $G^{\mathsf{rel}}_{u,v}$.*

Proof. Let π be an edit path from the start to the end vertex in the relevant edit graph. We construct an ordered edit sequence ρ by adding one operation for each arc with non-zero weight encountered on π. Since π has weight k, there are k non-zero arcs corresponding to k operations. Let p_{j_i} be the source and p_{j_i+1} the target vertex of the arc corresponding to the i-th operation. Let the row and column numbers of p_{j_i} be r and c. The path to p_{j_i} has weight $i - 1$, so we have $\mathrm{d}(\mathrm{pref}_c(u), \mathrm{pref}_r(v)) = i - 1$. The first $i - 1$ operations transform $\mathrm{pref}_c(u)$ to $\mathrm{pref}_r(v)$. The position of the i-th operation is $\mathrm{pos}\,(op_i) = r + 1$, since it transforms $\mathrm{pref}_{c+1}(u)$ to $\mathrm{pref}_{r+1}(v)$ (a substitution), $\mathrm{pref}_{c+1}(u)$ to $\mathrm{pref}_r(v)$ (a deletion), or $\mathrm{pref}_c(u)$ to $\mathrm{pref}_{r+1}(v)$ (an insertion). The row numbers on the path are strictly increasing except for the case of two deletions following immediately one after another. But, for two deletions op_i and op_{i+1} occurring directly in a row, we have the same positions $\mathrm{pos}\,(op_i) = \mathrm{pos}\,(op_{i+1})$. Therefore, the ordered edit sequence derived from the path is unique.

By induction on the number of operations, we show that each ordered edit sequence ρ corresponds uniquely to a path in the relevant edit graph. The start vertex corresponds surely to the edit sequence with no operations. Assume that we have found a unique path for the first i operations leading to a vertex p with row and column numbers r and c in the relevant edit graph such that the weight of its predecessor in the path is smaller than i (or p is the start vertex) and the first i operations transform $\mathrm{pref}_c(u)$ to $\mathrm{pref}_r(v)$. Thus, vertex p must be labeled with the weight i which is optimal and $\mathrm{d}(\mathrm{pref}_c(u), \mathrm{pref}_r(v)) = i$. The position of the i-th operation (if $i > 0$) is either r for a substitution or an insertion, or it is $r + 1$ for a deletion. Let the position of op_{i+1} be $\mathrm{pos}\,(op_{i+1}) = r'$ (i.e., we either replace or delete the r'-th character, or we insert another character in its place). Note that $r' \geq r + 1$ because we have $r' \geq r + 1$ if op_{i+1} is a deletion and $r' > r$ (hence $r' \geq r + 1$) if op_{i+1} is a substitution or an insertion. Since we have $\mathrm{d}(\mathrm{pref}_c(u), \mathrm{pref}_r(v)) = i$ and $\mathrm{pos}\,(op_{i+1}) = r'$, the intermediate substrings must be equal: $u[c..c + r' - 1 - r] = v[r..r' - 1]$. Thus, we must have zero-weight arcs from the vertex p to a vertex q with row and column numbers $r' - 1$ and $c + r' - 1 - r$. The weight of q is optimal because the weight of p is optimal by induction hypothesis. The vertex q represents the prefixes $\mathrm{pref}_{r'-1}(v)$ $\mathrm{pref}_{c+r'-1-r}(u)$ which have distance i. With the next operation, the first $i + 1$ operations transform $\mathrm{pref}_d(u)$ into $\mathrm{pref}_s(v)$, where $d = c + r' - r$ and $s = r' - 1$ for a deletion, $d = c + r' - r$ and $s = r'$ for a substitution, or $d = c + r' - 1 - r$ and $s = r'$ for an insertion. From q there is an arc corresponding to the next operation to a vertex p' with row and column numbers s and d: Each arc adds at most weight one. The path to p' is therefore optimal because the existence of a path to p' with less weight would prove the existence of an ordered edit sequence with fewer operations for the two prefixes. Thus, we could transform $\mathrm{pref}_d(u)$ into $\mathrm{pref}_s(v)$ with less than $i + 1$ and $\mathrm{suff}_{d+1}(u)$ into $\mathrm{suff}_{s+1}(v)$ with $k - i - 1$ operations. This, would be a contradiction to $\mathrm{d}(u, v) = k$.

Note that the position of the i-th operation derived from the edit path was $\mathrm{pos}\,(op_i) = r + 1$, where

r was the row number of the source vertex of the arc representing op_i in the first construction. Thus, if the operations have positions $r_1 + 1, \ldots, r_k + 1$, then the row numbers of the source vertices of non-zero weight arcs are $r_1, \ldots, r_k$. In the second construction, we derived the existence of a vertex at the start of an arc representing the $(i + 1)$-th operation with row and column numbers $r' - 1$ and $c + r' - 1 - r$, where r' was the position of the $(i + 1)$-th operation. Thus, if the non-zero weight arcs on the edit path start at row numbers $r_1, \ldots, r_k$, then the operations have positions $r_1 + 1, \ldots, r_k + 1$. As a result, we have a one-to-one mapping. □

Now, we can make the desired connection from the sequence of operations to the k-minimal prefix lengths.

Lemma 2.9 (Edit Stages of Ordered Edit Sequences). *Let $u, v \in \Sigma^*$ be such that $\mathrm{d}(u, v) = k$. Let $\rho(u, v) = (op_1, \ldots, op_k)$ be an ordered edit sequence and let $u(i) = op_i(\cdots op_1(u) \cdots)$ be the i-th edit stage. If we have $\mathrm{minpref}_{i,u(i)}(u) > h + 1$, then there exists an $j > h$ with $\mathrm{pref}_j(v) = \mathrm{pref}_j(u(i - 1))$.*

Proof. By Lemma 2.8, there is a unique path $\pi(u, v)$ in the relevant edit graph corresponding to the sequence $\rho(u, v)$. The same holds for any subsequence $\rho_i(u, u(i)) = (op_1, \ldots, op_i)$. Since there are no more operations when transforming u into $u(i)$, the remaining path which is not identical with the path for $\rho(u, v)$ must be made up of zero-weight arcs. Let p be the source vertex of the arc for the i-th operation on the edit path in the relevant edit graph $G^{\mathsf{rel}}_{u,u(i)}$. The vertex p must have weight $i - 1$. Let q be the target vertex of the arc for the $(i - 1)$-th operation. Up until vertex q, the edit paths in the relevant edit graphs $G^{\mathsf{rel}}_{u,u(i)}$ and $G^{\mathsf{rel}}_{u,u(i-1)}$ are equal. Thus, they are equal up to vertex p because there are only zero-weight arcs between q and p in the relevant edit graphs transforming u into $u(i)$. Furthermore, the same path is also found in the relevant edit graph $G^{\mathsf{rel}}_{u,v}$. Let the row and column numbers of p be r and c. Thus, p represents the prefixes $\mathrm{pref}_c(u)$ and

$$\mathrm{pref}_r(v) = \mathrm{pref}_r(u(i)) = \mathrm{pref}_r(u(i - 1)) \ . \tag{4}$$

Let p' be the next vertex in the edit path following p and let p' have row and column numbers r' and c'. Since p' has weight i, the distance between the represented prefixes is $\mathrm{d}(\mathrm{pref}_{r'}(u(i)), \mathrm{pref}_{c'}(u)) = i$. Hence, $\mathrm{minpref}_{i,u(i)}(u) \leq r'$. Since $r' \leq r + 1$, we find that

$$h + 1 < \mathrm{minpref}_{i,u(i)}(u) \leq r' \leq r + 1 \ , \tag{5}$$

and, thus, $r > h$. Equations (4) and (5) prove our claim. □

When looking at the edit graph, one can also see how it is possible to compute the edit distance in time $O(mk)$ for a pattern of length m and k errors. Since each vertical or horizontal edge increases the number of errors by one, there can be at most k such edges in any edit path from the start to the end vertex. Therefore, we only have to consider $2k + 1$ diagonals of length m. We call this a *k-bounded computation of the edit distance* (see also [Ukk85]).

3 Main Indexing Data Structure

In each of our indexing problems, in order to answer a query we have to find prefixes of strings in S that match the search pattern w with k errors. Assume that $S \subset \Sigma^*$ and $w \in \Sigma^*$ are given. We call

$s \in S$ a *k-error length-l occurrence of* w if $\mathrm{d}(\mathrm{pref}_l(s), w) = k$. We then also say that w matches s up to length l with k errors. We will sometimes omit the length l if it is either clear from context or irrelevant. To solve the indexing problems defined in Section 2.4 in optimal time, we need to be able to find all d-error occurrences of the search pattern w in time $O(|w|)$. For that purpose, we will define error trees so that leaves in certain subtrees contain the possible matches. From these, we select the desired output using range queries.

On an abstract level, the basic idea of the index for d errors is the following: A single string $s \in S$ can be compared to the pattern w to check whether w matches a prefix of s with at most d errors in time $O(d \cdot |w|)$ as described in Section 2.6. On the other hand, a precomputed index of all strings within edit distance d of S (e.g., a trie containing all strings r for which r matches a prefix of a string $s \in S$ with at most d errors) allows a linear look-up time. Both methods are relatively useless on their own: The index would be prohibitively large while comparing the pattern to each string would take too long. Therefore, we take a balanced approach by precomputing some of the neighboring strings and directly computing the distance for others. In particular, we inductively define sets of strings $W_0, \ldots, W_d$ where, for $0 \leq i \leq d$, the set W_i consists of strings with edit distance i to some string in S. We begin with $S = W_0$. Parameters $h_0, \ldots, h_d$ are used to control the precomputing. As an example, W_1 consists of all strings $r = u'v$ such that $\mathrm{d}(r, s) = 1$ for some $s = uv \in W_0$ and $|u'| \leq h_0 + 1$, i.e., we apply all possible edit operations to strings $s \in S$ that result in a modified prefix of length $h_0 + 1$. The partitioning into $d + 1$ sets allows searching with different error bounds up to d.

3.1 Intuition

The intuitive idea is as follows. If the search pattern w matches a prefix s of some string in S with no errors, we find it in the trie $\mathcal{T}$ for S. If w matches s with one error, then there are two possibilities depending on the position of the error: Either the error lies above the height h_0 of the trie $\mathcal{T}$, or below. If it lies above, we find a prefix of w in the trie reaching a leaf edge. Thus, we can simply check the single string corresponding to the leaf by a k-bounded computation of the edit distance (with $k = 1$) in time $O(|w|)$. Otherwise, the error lies below the height of the trie and is covered by precomputing. For this case, we compute all strings r that are within distance one to a string in S such that the position of the error is in a prefix of length $h_0 + 1$ of r. At each position we can make at most $2|\Sigma|$ errors ($|\Sigma|$ insertions, $|\Sigma| - 1$ substitutions, one deletion). Thus, for each string S we generate $O(h_0)$ new strings (remember that we consider $|\Sigma|$ to be constant). The new strings are inserted in a trie $\mathcal{T}'$ with height h_1. For the second case, we find all matches of w in the subtree $\mathcal{T}'_w$.[3]

We extend this scheme to two errors as follows. Assume w matches a prefix $s \in_{\mathsf{pref}} S$ of a string in the base set with two errors. There are three cases depending on the positions of the errors (of an ordered edit script). If the first error occurs above the height h_0 of the trie $\mathcal{T}$, then we find a prefix of w in $\mathcal{T}$ leading to a leaf edge. Again, we can check the single string corresponding to this edge in time $O(|w|)$ as above. If the first error occurs below the height $h_0 + 1$, then we have inserted a string r into the trie $\mathcal{T}'$ that reflects this first error. We are left with two cases: Either the second error occurs above the height h_1 of $\mathcal{T}'$ or below. In the first case, there is a leaf representing a single string[4] which we can check in time $O(|w|)$ as above. Finally, for the case that the second error occurs below h_1, we generate $O(h_1)$ new strings for each string in $\mathcal{T}'$ in the same manner as before. The new strings are

[3] Note, though, that some leaves in $\mathcal{T}'_w$ may not be one error matches of w because the error may be after the position $|w|$. We will employ range queries to select the correct subset of leaves.

[4] We have to relax this later so that a leaf in the trie for i errors might represent $2i + 1$ strings.

inserted in a trie $\mathcal{T}''$ with height h_2. Again, we find all matches of w in the subtree $\mathcal{T}''_w$.

The idea can be carried on to any (constant) number of errors d, each time making a case distinction on the position of the next error.

3.2 Definition of the Basic Data Structure

We now rigidly lay out the idea. There are some more obstacles that we have to overcome when generating new strings from a set W_i: We have to avoid generating a string by doing a reverse error, i.e., undoing an error from a previous step. Also, to ease representation, we allow all errors on prefixes of strings that lead to a new prefix of maximal length $h_{i-1} + 1$. This, in particular, also captures all operations with positions up to $h_{i-1} + 1$. (See the proof of Lemma 2.8: The position of an operation corresponds to one plus the row number of the source vertex in the edit graph. The resulting string has length at most one plus this row number.) The key property of the following inductively defined sets $W_0, \ldots, W_d$ is that we have at least one error before $h_0 + 1$, two errors before $h_1 + 1$, and so on until we have d errors before $h_{d-1} + 1$ in W_d.

Definition 3.1 (Error Sets). *For $0 \leq i \leq d$, the error set W_i is defined inductively. The first error set W_0 is defined by $W_0 = S$. The other sets are defined by the recursive equation*

$$W_i = \Gamma_{h_{i-1}}(W_{i-1}) \cap S_i \ \ , \tag{6}$$

where Γ_l is an operator on sets of strings defined for $A \subset \Sigma^*$ by

$$\Gamma_l = \{op(u)v \mid uv \in A, |op(u)| \leq l + 1, op \in \{\mathrm{del}, \mathrm{ins}, \mathrm{sub}\}\} \ \ , \tag{7}$$

the set S_i is defined as

$$S_i = \{r \mid \textit{there exists } s \in S \textit{ such that } \mathrm{d}\,(s, r) = i\} \tag{8}$$

(i.e., the strings that are within distance i of the string in S), and h_i are integer parameters (that may depend on W_i).

We say that $r \in W_i$ *stems from* $s \in S$ if $r = op_i(\cdots op_1(s) \cdots)$. If the base set S contains suffixes of a string u, then the same string $r \in W_i$ may stem from multiple strings $s, t \in S$. For example, if we have $t = av$ and $s = v$, then bs has distance one to both, t and s. When generating W_i from W_{i-1} it is therefore necessary to eliminate duplicates. On the other hand, we do not want to eliminate duplicates generated from independent strings. Therefore, we introduce sentinels and extend the distance function d. Each string is appended with a different sentinel, and the distance function is extended in such a way that the cost of applying an operation to a sentinel is at least $2d + 1$. To avoid the introduction of a sentinel for each string, we can even use logical sentinels: We define the distance between the sentinel and itself to be either zero if the sentinels come from the same string, or to be at least $2d + 1$ if the sentinels come from two different strings.

With the scheme we just described, we can eliminate duplicates by finding all those newly generated strings t and s that stem from the same string or from different suffixes of the same string $u \in W_{i-1}$, have the same length, and a common prefix. The following property will be useful to devise an efficient algorithm in the next section.

Lemma 3.2 (Error Positions in Error Sets). *Assume that the string $r \in W_i$ stems from the string $u \in S$ by a sequence of operations $op_1, \ldots, op_i$. If the parameters h_i are strictly increasing for $0 \le i \le d$, then $\mathrm{suff}_{h_{i-1}+2}(r) = \mathrm{suff}_{h_{i-1}+2}(u)$ and $\mathrm{suff}_{h_i+1}(r) = \mathrm{suff}_{h_i+1}(u)$.*

Proof. We claim that any changed, inserted, or deleted character in r occurs in a prefix of length $h_{i-1} + 1 \le h_i$ of r. By assumption, $r = op_i(\cdots op_1(u) \cdots)$. Let $r(j) = op_j(\cdots op_1(u) \cdots)$ be the edit stages. In the j-th step, the position of the applied operation is $\mathrm{pos}\,(op_j)$, thus op_j has changed, or inserted the $\mathrm{pos}\,(op_j)$-th character in string $r(j)$ or deleted a character from position $\mathrm{pos}\,(op_j)$ of the string $r(j-1)$. We denote this position by p_j (for $r(j-1)$ we have $p_j = \mathrm{pos}\,(op_j)$, but the position changes with respect to later stages). Consecutive operations may influence p_j, i.e., a deletion can decrease and an insertion increase p_j from $r(j)$ to $r(j+1)$ by one. Thus, after $i - j$ operations, $p_j \le \mathrm{pos}\,(op_j) + i - j$. By Definition 3.1, we have $\mathrm{pos}\,(op_j) \le h_{j-1} + 1$. Therefore, in step i, $p_j \le h_{j-1} + 1 + i - j$. Since the h_i are strictly increasing, we have $h_{j-1} + 1 \le h_j \le h_{j+1} - 1 \le \cdots \le h_{i-1} + 1 - i + j \le h_i - i + j$. As a result, for any position in r, where a character was changed, we have $p_j \le h_{i-1} + 1 \le h_i$. □

To search the error sets efficiently, and to facilitate the elimination of duplicates, we use weak tries, which we call *error trees*.

Definition 3.3 (Error Trees). *The i-th error tree $\mathrm{et}_i(S)$ is defined as the weak trie $\mathcal{W}_{h_i}(W_i)$. Each leaf p of $\mathrm{et}_i(S)$ is labeled (id_s, l), for each $s \in S$ where id_s is an identifier for s and $l = \mathrm{minpref}_{i,\mathrm{path}(p)}(s)$.*

To capture the intuition first, it is easier to assume that we set $h_i = \mathrm{maxpref}(W_i)$, i.e., the maximal length of a common prefix of any two strings in W_i. For this case the error trees become compact tries.

A leaf may be labeled multiple times if it represents different strings in S (but only once per string). For i errors, the number of suffixes of a string t that may be represented by a leaf p is bounded by $2i + 1$: The leaf p represents a path $\mathrm{path}(p) = u$ and any string matching u with i errors has to have a length between $|u| - i$ and $|u| + i$. We assumed that $i \le d$ is constant, thus a leaf has at most constantly many labels. The labels can easily be computed while eliminating duplicates during the generation of the set W_i.

3.3 Construction and Size

To allow d errors, we build the $d + 1$ error trees $\mathrm{et}_0(S), \ldots, \mathrm{et}_d(S)$. We start constructing the trees from the given base set S. Each element $r \in W_i$ is implemented by a reference to a string $s \in S$ and an annotation of an ordered edit sequence that transfers $u = \mathrm{pref}_{l+|s-|r||}(s)$ into $v = \mathrm{pref}_l(r)$ for $l = \mathrm{minpref}_{i,r}(s)$. The i-th error set is implicitly represented by the i-th error tree. We build the i-th error tree by generating new strings with one additional error from strings in W_{i-1}. Since we annotated an ordered edit sequence to each string, we can easily avoid to undo a previous operation with the new one. The details are given in the next lemma.

Lemma 3.4 (Construction and Size of W_i and $\mathrm{et}_i(S)$). *For $0 \le i \le d$, assume that the parameters h_i are strictly increasing, i.e., $h_i > h_{i-1}$, and let $n_i = |W_{i-1}|$. The set W_i can be constructed from the set W_{i-1} in time $O(n_{i-1}h_{i-1}h_i)$ and space $O(n_{i-1}h_{i-1})$ yielding the error tree $\mathrm{et}_i(S)$ as a byproduct. The i-th error tree $\mathrm{et}_i(S)$ has size $O(|S|h_0 \cdots h_{i-1})$.*

Proof. We first prove the space bound. For each string r in W_i, there exists at least one string r' in W_{i-1} such that $r = op(r')$ for some edit operation. For string r' in W_{i-1} there can be at most $2|\Sigma|(h_{i-1}+2)$ strings in W_i: Set $u' = \mathrm{pref}_{h_{i-1}+1}(r')$, then we can apply at most $|\Sigma|(h_{i-1}+1)$ insertions, $(|\Sigma|-1)(h_{i-1}+1)$ substitutions, and $(h_{i-1}+2)$ deletions. Hence, $|W_i| \leq 2|\Sigma|(h_{i-1}+2)|W_{i-1}|$.

Let W_i' be the multi-set of all strings constructed from W_{i-1} by applying all possible edit operations that result in modified prefixes of length $h_{i-1}+1$. We have to avoid undoing an earlier operation. This can be done by comparing the new operation with the annotated edit sequence. Note that we may construct the same string from multiple edit sequences (also including the ordered edit sequence). To construct W_i from W_i', we have to eliminate the duplicates.

By Lemma 3.2, if the string $r \in W_i'$ stems from the string $u \in S$, then $\mathrm{suff}_{h_i+1}(r) = \mathrm{suff}_{h_i+1}(u)$. If $r, s \in W_i'$ are equal, then they must either stem both from the same string $u \in S$, or they are both suffixes of the same string u. Since there are no errors after h_i, $\mathrm{suff}_{h_i+1}(r) = \mathrm{suff}_{h_i+1}(s) = \mathrm{suff}_{h_i+1}(u)$. Note that $h_i + 1 - i \leq |\mathrm{suff}_{h_i+1}(u)| \leq h_i + 1 + i$, so there can be at most $2i+1$ different suffixes for any string u.

To eliminate duplicates, we build an h_i-weak trie by naively inserting all strings from W_i'. Let n be the number of independent strings that were used to build the base set S, i.e., all suffixes of one string count for one. Obviously, $n \leq |S|$. We create $n \cdot (2d+1)$ buckets and sort all leaves hanging from a node p into these in linear time. All leaves in one bucket represent the same string. For suffixes, we select one leaf and all different labels thereby also determining the k-minimal prefix length. For buckets of other strings we just select the one leaf with the label representing the k-minimal prefix. After eliminating the surplus leaves, the weak trie becomes $\mathrm{et}_i(S)$.

Building the h_i-weak trie for W_i' takes time $O(h_i|W_i'|) = O(n_{i-1}h_{i-1}h_i)$, and eliminating the duplicates can be done in time linear in the number of strings in W_i'.

The size of the i-th error tree is linear in the number of leaf labels and thus bounded by the size of W_i. Iterating $|W_i| = O(h_{i-1}|W_{i-1}|)$ leads to $O(|S|h_0 \cdots h_{i-1})$. □

We choose the parameters h_i either as $h_i = \mathrm{maxpref}(W_i)$ or as $h_i = h + i$ for some fixed value h that we specify later. By both choices we satisfy the assumptions of Lemma 3.4 as shown by the following lemma.

Lemma 3.5 (Increasing Common Prefix of Error Sets). *For $0 \leq i \leq d$, let $h_i = \mathrm{maxpref}(W_i)$ be the maximal prefix of any two strings in the i-th error set, then $h_i > h_{i-1}$.*

Proof. We prove by induction. Let r and s be two strings in W_{i-1} with a maximal prefix $\mathrm{pref}_{h_{i-1}}(r) = \mathrm{pref}_{h_{i-1}}(s) = u$ for some $u \in \Sigma^{h_{i-1}}$. Since r and s are not identical, we have $r = uav$ and $s = ubv'$ for some strings $v, v' \in \Sigma^*$ and $a, b \in \Sigma$. We have $|\Sigma| \leq 2$, therefore, $\Gamma(W_{i-1})$ contains at least ubv, $uaav$, $ubav$, uv, uav', $uabv'$, $ubbv'$, and uv'. By Lemma 3.2, for any string $r \in W_{i-1}$ that stems from $t \in S$, we have $\mathrm{suff}_{h_{i-1}+1}(r) = \mathrm{suff}_{h_{i-1}+1}(t)$, thus no character at a position greater or equal to $h_{i-1}+1$ can have been changed by a previous operation. Thus, applying any operation at $h_{i-1}+1$ creates a new string that has distance i to some string in S. Both $ubav$ and $ubbv'$ were created by inserting a character at $h_{i-1}+1$, so they do not undo an operation and belong to W_i. They both have a common prefix of length at least $h_{i-1}+1 > h_{i-1}$. Thus, we find that $h_i > h_{i-1}$. □

For $h_i = \mathrm{maxpref}(W_i)$ we do not need to use the buckets as described in the proof of Lemma 3.4 but we can create the error trees more easily. By assumption, two strings are either completely identical, or they have a common prefix of size at most h_i. On the other hand, no two strings have a common prefix of more than h_i, thus there can be at most one bucket below h_i anyway. If two strings r and s are identical, they must stem from suffixes of the same string $u \in S$ and the suffixes $\mathrm{suff}_{h_{i+1}}(r)$ and $\mathrm{suff}_{h_{i+1}}(s)$ must be equal. Since s and r are implemented by pointers to strings in S we can easily check whether they reference the same suffix of the same string during the process of insertion and before comparing at a character level or not.

3.4 Main Properties of the Data Structure

The sequence of sets W_i simulates the successive application of d edit operations on strings of S, where the position of the i-th operation is limited to be smaller than $h_{i-1} + 1$. Before describing the search algorithms, we take a look at some key properties of the error sets that show the correctness of our approach.

Lemma 3.6 (Existence of Matches). *Let $w \in \Sigma^*$, $s \in S$, and $t \sqsubseteq_{\mathsf{pref}} s$ be such that $\mathrm{d}(w,t) = i$. Let $\rho(w,t)$ be an ordered edit sequence for $w = op_i(\cdots op_1(t)\cdots)$. For $0 \le j \le i$, let $t(j) = op_j(\cdots op_1(t)\cdots)$ be the j-th edit stage. If for all $1 \le j \le i$ we have*

$$\mathrm{minpref}_{j,t(j)}(t) \le h_{j-1} + 1 \ , \tag{9}$$

then there exists a string $r \in W_i$ with

$$w \sqsubseteq_{\mathsf{pref}} r, \quad r = op_i(\cdots op_1(s)\cdots), \quad \textit{and} \quad l = \mathrm{minpref}_{i,r}(s) \ . \tag{10}$$

Proof. We prove by induction on i. For $i = 0$, $W_0 = S$ and the claim is obviously true since $w = t \sqsubseteq_{\mathsf{pref}} s$ in that case.

Assume the claim is true for all $j \le i-1$. Let $r = op_i(\cdots op_1(s)\cdots)$. Since $\mathrm{d}(r,s) = i$, we have to show that there exist $r' \in W_{i-1}$ and strings $u, u', v \in \Sigma^*$, with $r = uv$, $r' = u'v$, $\mathrm{d}(u,u') = 1$, and $|u'| \le h_{i-1} + 1$. Then $r \in W_i$ by Definition 3.1.

Set $l = \mathrm{minpref}_{i,r}(s)$. Since w and t are prefixes of r and s which already have distance i, we have $l = \mathrm{minpref}_{i,w}(t)$ and $l \le h_{i-1} + 1$ by equation (9). Set $u = \mathrm{pref}_{l+|s|-|r|}(s)$, $u' = \mathrm{pref}_l(r)$, and $v = \mathrm{suff}_{l+1}(r)$. By Definition 2.7, $r = u'v$, $s = uv$, $\mathrm{d}(u,u') = i$, and $u' = op_i(\cdots op_1(u)\cdots)$. Set $u'' = op_{i-1}(\cdots op_1(u)\cdots)$ and $r' = u''v$, then $r' = op_{i-1}(\cdots op_1(s)\cdots)$. We are finished if we can show that $r' \in W_{i-1}$ because $|u'| = l \le h_{i-1} + 1$ and $\mathrm{d}(u', u'') = 1$.

Let $w' = op_{i-1}(\cdots op_1(t)\cdots)$, then $\mathrm{d}(w',t) = i-1$. Since for all $1 \le j \le i-1$, we have $\mathrm{minpref}_{j,t(j)}(t) \le h_{j-1} + 1$ we can apply the induction hypothesis and find that there exists a string $\hat{r} = op_{i-1}(\cdots op_1(s)\cdots)$ in W_{i-1} with $w' \sqsubseteq_{\mathsf{pref}} \hat{r}$ and $l' = \mathrm{minpref}_{i-1,\hat{r}}(s)$. Since $\hat{r} = r'$ this proves our claim. □

When we translate this to error trees, we find that, given a pattern w, the i-error length-l occurrence s of w corresponding to a leaf labeled by (id_s, l) can be found in $\mathrm{et}_i(S)_w$. Unfortunately, not all leaves in a subtree represent such an occurrence. The following lemma gives a criterion for selecting the leaves (the errors must appear before w, i.e., if $l \le |w|$).

Lemma 3.7 (Occurrences Leading to Matches). *For $r \in W_i$, let $w \sqsubseteq_{\mathsf{pref}} r$ be some prefix of r and let $s \in S$ be a string corresponding to r such that $l = \mathrm{minpref}_{i,r}(s)$. There exists a prefix $t \sqsubseteq_{\mathsf{pref}} s$ such that $\mathrm{d}(t,w) = i$ and $\mathrm{suff}_{|w|+1}(r) = \mathrm{suff}_{|t|+1}(s)$ if and only if $|w| \ge l$.*

Proof. If $|w| \geq l$, then there are strings $u, v \in \Sigma^*$ with $w = uv$ and $u = \mathrm{pref}_l(w)$. Since w is a prefix of r, $r = wx = uvx$. By Definition 2.7, there also exists a prefix $u' \sqsubseteq_{\mathsf{pref}} s$ with $s = u'vx$ and $\mathrm{d}(u', u) = i$. Hence, $t = u'v \sqsubseteq_{\mathsf{pref}} s$, $\mathrm{d}(w, t) = \mathrm{d}(uv, u'v) = i$, and $x = \mathrm{suff}_{|w|+1}(r) = \mathrm{suff}_{|t|+1}(s)$.

Conversely, assume that $|w| < l$ and there exists a prefix $t \sqsubseteq_{\mathsf{pref}} s$ with $\mathrm{d}(t, w) = i$ and $\mathrm{suff}_{|w|+1}(r) = \mathrm{suff}_{|t|+1}(s)$. Set $m = |w|$ and recall that $w = \mathrm{pref}_m(r)$. Then $s = t{\cdot}\mathrm{suff}_{|w|+1}(r)$ and, thus, $t = \mathrm{pref}_{|s|-|r|+m}(s)$. Then $\mathrm{d}(\mathrm{pref}_m(r), \mathrm{pref}_{m+|s|-|r|}(s)) = i$ and $\mathrm{suff}_{m+1}(r) = \mathrm{suff}_{m+1+|s|-|r|}(s)$, i.e., m is a candidate for $\mathrm{minpref}_{i,r}(s)$, which is a contradiction to $m < l$. □

In the error tree, this means that we have an i-error occurrence of w for a leaf p in $\mathrm{et}_i(S)_w$ with path $\mathrm{path}(p) = r$ if and only if p is labeled (id_s, l) with $|w| \geq l$. Finally, there is a dichotomy which directly implies an efficient search algorithm.

Lemma 3.8 (Occurrence Properties). *Assume w matches $t \sqsubseteq_{\mathsf{pref}} s$ for $s \in S$ with i errors, i.e., $\mathrm{d}(w, t) = i$. Let $\rho = (op_1, \ldots, op_i)$ be an ordered edit sequence such that $w = op_i(op_{i-1}(\cdots op_1(t) \cdots))$. There are two mutually exclusive cases,*

1. *either $w \in_{\mathsf{pref}} W_i$, or*
2. *there exists at least one $0 \leq j \leq i$, such that we find $r \in W_j$ with $r = op_j(\cdots op_1(s) \cdots)$ and for some $w' \sqsubseteq_{\mathsf{pref}} r$ we have $w' = \mathrm{pref}_l(w)$ for some $l > h_j$.*

Proof. Set $t(j) = op_j(\cdots op_1(t) \cdots)$. Assume that there exists no j such that $\mathrm{minpref}_{j,t(j)}(t) > h_{j-1} + 1$. Then $w \in_{\mathsf{pref}} W_i$ by Lemma 3.6.

Otherwise, let j be the smallest index such that $\mathrm{minpref}_{j+1,t(j+1)}(t) > h_j + 1$. By Lemma 3.6, there exists a string $r \in W_j$ with $t(j) \sqsubseteq_{\mathsf{pref}} r$. By Lemma 2.9, there exists a prefix $w' = \mathrm{pref}_l(w) = \mathrm{pref}_l(t(j))$ with $l > h_j$, thus $w' \sqsubseteq_{\mathsf{pref}} r$. □

When searching for a pattern w, assume that either $h_i > \mathrm{maxpref}(W_i)$ or $|w| \leq h_i$ for all i. The last lemma applied to error trees shows that if w matches a string $t \sqsubseteq_{\mathsf{pref}} s$ for some $s \in S$ with exactly i errors, then the following dichotomy occurs.

Case A Either w can be matched completely in the i-th error tree $\mathrm{et}_i(S)$ and a leaf p labeled (id_s, l) can be found in $\mathrm{et}_i(S)_w$.

Case B Or a prefix $w' \sqsubseteq_{\mathsf{pref}} w$ of length $|w'| > h_j$ is found in $\mathrm{et}_j(S)$ and $\mathrm{et}_j(S)_{w'}$ contains a leaf p with label (id_s, l).

3.5 Search Algorithms

Lemmas 3.7 and 3.8 directly imply a search algorithm along the case distinction made above. Recall that we can build the index efficiently if we choose the parameters h_i either as $h_i = \mathrm{maxpref}(W_i)$ or as $h_i = h + i$ for some fixed value h. The index supports searches if either $h_i = \mathrm{maxpref}(W_i)$ or the length of the search pattern w is bounded by $|w| \leq h$. For these parameters, we can check all prefixes $t \in_{\mathsf{pref}} S$ for which Case B applies in time $O(|w|)$: If $|w| \leq h$, then we can never have $|w'| > h_i$ for a prefix $w' \sqsubseteq_{\mathsf{pref}} w$ and so the case never applies. Otherwise, we have $h_i = \mathrm{maxpref}(W_i)$. In this case, the error trees become tries and so there is at most one leaf in $\mathrm{et}_i(S)_{w'}$ if the length of the prefix $w' \sqsubseteq_{\mathsf{pref}} w$ is greater than h_i. Each leaf can have at most $2d+1$ labels corresponding to at most $2d+1$ strings from S. We compute the edit distance of w to every prefix of each strings in time $O(|w|)$ with a d-bounded computation of the edit distance (see Section 2.6). There can by at most $2d+1$ prefixes that match w, thus, from all $d+1$ error trees, we have at most $d(2d+1)^2$ strings in total for which we must check the problem specific conditions for reporting them. As a result, we get the following lemma.

Lemma 3.9 (Search Time for Case B). *If d is constant and either $h_i = \operatorname{maxpref}(W_i)$ or the length of the search pattern w is bounded by $|w| \leq h \leq \min_i h_i$, then the total search time spent for Case B is $O(|w|)$.*

Case A is more difficult because we have to avoid reporting occurrences multiple times. A string with errors above $|w|$ can occur multiple times in $\mathrm{et}_i(S)_w$. Since any string t matching the pattern w with d or less errors has length $|w| - d \leq |t| \leq |w| + d$, we can eliminate duplicate reports using $|S|(2d+1)$ buckets if necessary. The main issue is to restrict the number of reported elements for each error tree i. If we can ensure that no output candidate is reported twice in each error tree, then the total work for reporting the outputs is linear in the number of outputs.

The strings in the base set S can be either independent or they are suffixes of a string u (also called document). Recalling the definition for the base set made in Section 2.4, we have to support four different **types** of selections:

1. For the version (a) of problems d-ATI, d-ADI, and d-ADCI we want to report each prefix t of any string $s \in S$ that matches the pattern w with at most d errors.

2. For the version (b) of problems d-ATI, d-ADI, and d-ADCI we want to report each string $s \in S$ for which a prefix $t \sqsubseteq_{\mathsf{pref}} s$ matches the pattern w with at most d errors.

3. For the version (c) of problem d-ADI we want to report each string $s \in S$ which matches the pattern w with at most d errors.

4. For the version (c) of problem d-ADCI we want to report each document u if a prefix $t \sqsubseteq_{\mathsf{pref}} s$ of a suffix $s \in S$ of u matches the pattern w with at most d errors.

The basic task is to match the pattern in each of the $d+1$ error trees. If the complete pattern could be matched, we are in Case A. To support the selection of the different types, we create additional arrays for each error tree. Let n_i be the number of leaf labels in the i-th error tree $\mathrm{et}_i(S)$. First, we create an array A_i of size n_i that contains each leaf label and a pointer to its leaf in the order encountered by a depth first traversal of the error tree. For example, if the weak tree in Figure 1(c) were an error tree, we would first store the labels of the node 4, then all labels of the node 6, and so on until we have stored all labels node 21 at the end of A_i. The order of the depth first traversal can be arbitrary but must be fixed. Each node p of the error tree is annotated by the leftmost index $\mathrm{left}(p)$ and the rightmost index $\mathrm{right}(p)$ in A_i containing a leaf label from the subtree rooted under p. For a virtual node q, $\mathrm{left}(q)$ and $\mathrm{right}(q)$ are taken from the next non-virtual node below q.

To support the selection of results, we create an additional array B_i of the same size. Depending on the type, B_i contains an integer value used to select the corresponding leaf label using range minimum queries.

By Lemma 3.7, for reports of Type 1, we have to select the strings corresponding to those labels for which the minimal prefix value l stored in the label is smaller than the length of the pattern. This is achieved by setting $B_i[j]$ to l if $A_i[j]$ contains the leaf label (id_s, l). Let p be the (virtual) node corresponding to the location of the pattern w in the error tree $\mathrm{et}_i(S)$. A bounded value range (BVR) query $(\mathrm{left}(p), \mathrm{right}(p))$ with bound $|w|$ on B_i yields the indices of labels in A_i for which $l \leq |w|$ in linear time in the number of labels.

For the reports of Type 2, we just have to select the strings corresponding to matches. Observe that for any label (id_s, l) found in the subtree $\mathrm{et}_i(S)_w$ there is a prefix of s that matches w with at most i errors. Hence, we simply store in $B_i[j]$ an identifier for the string s if (id_s, l) is stored in $A_i[j]$. Let again p be the (virtual) node corresponding to the location of the pattern w in the error tree $\mathrm{et}_i(S)$. A colored range (CR) query $(\mathrm{left}(p), \mathrm{right}(p))$ on B_i yields the different string identifiers found in $\mathrm{et}_i(S)_w$.

The reports of Type 3 require the complete string to be matched by the pattern w. We have to take care of sentinels that we have added to the strings in S. Let p be the (virtual) node corresponding to the location of the pattern w in the error tree $\mathrm{et}_i(S)$. Since we added a different sentinel for each string in S, there is a match only if there is an outgoing edge labeled by a sentinel at p, and there is one such outgoing edge for each different string in S. Thus, we can report the matches directly from the error tree.

Finally, for reports of Type 4, we want to report the documents of which the strings in S are suffixes of. For this case, the string identifiers must contain a document number and we use the same approach as for Type 2, just storing document numbers in B_i.

Lemma 3.10 (Search Time for Case A). *If d is constant, we report occ outputs, and either $h_i = \mathrm{maxpref}(W_i)$, or the length of the search pattern w is bounded by $|w| \leq h \leq \min_i h_i$, then the total search time spent for Case A is $O(|w| + \mathrm{occ})$.*

Proof. Matching the pattern w in each of the d error trees takes time $O(|w|)$ because we never reach the part of the weak tries where the unique branching criterion does not hold. Thus, we find a single (virtual) node representing w. The range queries (or the tree traversal for Type 3) are performed in linear time in the number of outputs occ. Each output is generated at most $d+1$ times. Therefore, the total time is $O(|w| + \mathrm{occ})$. □

For the space we have the following obvious lemma.

Lemma 3.11 (Additional Preprocessing Time and Space for Case A). *The additional space and time needed for the range queries and the arrays for solving Case A is linear in the number of leaves of the errors trees.*

Proof. There are at most $2d + 1$ labels per leaf and so the size of the arrays generated is linear in the number of leaves. The time needed for the depth first traversals is also linear in the array sizes. Finally, the range queries are also prepared in time and space linear in the size of the arrays (see Section 2.5). □

4 Worst-Case Optimal Search-Time

When setting h_i to $\mathrm{maxpref}(W_i)$, our main indexing data structure already yields worst-case optimal search-time by Lemmas 3.9 and 3.10. What is left is to determine the size of the data structure and the time needed for preprocessing. Note that already for $i = 0$, $h_0 = \mathrm{maxpref}(W_0) = \mathrm{maxpref}(S)$ can be of size $\Omega(n)$ if S is the set of suffixes of a string of length n. For independent strings, the worst-case size of $h_0 = \mathrm{maxpref}(S)$ cannot be bounded at all in terms of $n = |S|$. Fortunately, the average size is much better and it occurs with high probability. In this section we derive the corresponding average-case bounds for $h_i = \mathrm{maxpref}(W_i)$. Together with Lemmas 3.4 and 3.11, this gives a bound on the total size and preprocessing time because they are all dominated by the size and preprocessing time needed for the d-th error tree:

Corollary 4.1 (Data Structure Size and Preprocessing Time). *Let n be the number of strings in the base set S. For constant d, the total size of the main data structures used is $O(n \cdot h_0 \cdot h_1 \cdot \dots \cdot h_{d-1})$ and the time for preprocessing is $O(n \cdot h_0 \cdot h_1 \cdot \dots \cdot h_{d-1} \cdot h_d)$.*

We show that, under the mixing model for stationary ergodic sources, the probability that h_i deviates significantly from $c \log n$ is exponentially small. Using this bound, we can also show that the expected value of h_i is $O(\log n)$.

Let $\{X_k\}_{k \geq 1}$ be a random sequence generated by a stationary and ergodic source. For $n < m$, let $\mathbb{F}_n^m$ be the σ-field generated by $\{X_k\}_{k=n}^m$ with $1 \leq n \leq m$. The source satisfies the *mixing condition* if there exist positive constants $c_1, c_2 \in \mathbb{R}$ and $d \in \mathbb{N}$ such that for all $1 \leq m \leq m + d \leq n$ and for all $\mathcal{A} \in \mathbb{F}_1^m$, $\mathcal{B} \in \mathbb{F}_{m+d}^n$, the inequality

$$c_1 \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \leq \Pr\{\mathcal{A} \cap \mathcal{B}\} \leq c_2 \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \tag{11}$$

holds. Note that this model encompasses the memoryless model and stationary and ergodic Markov chains. Under this model, the following limit (the Rény entropy of second order) exists

$$r_2 = \lim_{n \to \infty} \frac{-\ln\left(\sum_{w \in \Sigma^n} \left(\Pr\{w\}\right)^2\right)}{2n} \quad , \tag{12}$$

which can be proven by using sub-additivity [Pit85].

If $h_i = \mathrm{maxpref}(W_i)$ is greater than l, there must be two different string s and r in W_i such that $\mathrm{pref}_l(s) = \mathrm{pref}_l(r)$. We first prove that this implies the existence of an exact match between two substrings of length $\Omega(\frac{l}{i})$ of strings in S, then we bound the probability for this event. Note that, if S contains all suffixes of a string v, then S also contains v itself.

Lemma 4.2 (Length of Common or Repeated Substrings). *Let $W_0, \dots, W_d$ be the error sets by Definition 3.1. If there exists an i with $h_i \geq (2i+1)l$ for $l > 1$, then there exists a string v of length $|v| > l$ such that either v is a substring of two independent strings in S, or $v = u[j..j+l-1] = u[j'..j'+l-1]$ with $j \neq j'$ for some $u \in S$. Furthermore, for $l \geq 2$, both occurrences of v start in a prefix of length $2il$ of strings in S.*

Proof. We prove the claim by induction over i. For $i = 0$, the claim is naturally true because h_0 is the length of the longest prefix between two strings in S. This is either the longest repeated substring in a string u (if all suffixes of u were inserted into S), or the longest common prefix of two independent strings.

For the induction step, assume that for all $j < i$ we have $h_j < (2j+1)l$ and that $h_i \geq (2i+1)l$. Let $r, s \in W_i$ be two strings with a common prefix v of length $|v| = h_i \geq (2i+1)l$, thus $v = \mathrm{pref}_{|v|}(r) = \mathrm{pref}_{|v|}(s)$. Let $t^{(r)}, t^{(s)} \in S$ be the elements from the base set corresponding to r and s, i.e., $\mathrm{d}(t^{(r)}, r) = i$ and $\mathrm{d}(t^{(s)}, s) = i$. Recall that, by Lemmas 3.2 and 3.5, $\mathrm{suff}_{h_{i-1}+2}(r) = \mathrm{suff}_{h_{i-1}+2}(t)$ if $r \in W_i$ stems from $t \in S$. It follows that $t^{(r)}$ and $t^{(s)}$ share the same substring $w = t^{(r)}[h_{i-1}+2..h_i] = t^{(s)}[h_{i-1}+2..h_i]$ of length $|w| = h_i - h_{i-1} - 2 + 1 > (2i+1)l - (2(i-1)+1)l - 1 = 2l - 1$. Even if $t^{(r)}$

and $t^{(s)}$ are the same string u or suffixes of the same string u, then w cannot start at the same position in u: Assume for contradiction that $w = t^{(r)}[h_{i-1}+2..h_i] = t^{(s)}[h_{i-1}+2..h_i] = u[k..k+|w|]$, then $\mathrm{suff}_{h_{i-1}+2}(t^{(r)}) = \mathrm{suff}_k(u) = \mathrm{suff}_{h_{i-1}+2}(t^{(s)})$, so r and s do not branch at h_i. This is a contradiction.

The last claim follows from $w = t^{(r)}[h_{i-1}+2..h_i]$ and $h_{i-1} \leq (2i-1)l$. Thus, w starts before $(2i-1)l+2 \leq 2il$ for $l \geq 2$ in $t^{(r)}$ and likewise for $t^{(r)}$. □

For the analysis, we assume the mixing model introduced above. The intuition for the next theorem is as follows. The height of (compact) tries and suffix trees is bounded by $O(\log n)$ where n is the cardinality of the input set for tries or the size of the string for suffix trees (see, e.g., [AS92] or [Szp00]). When allowing an error on prefixes bounded by the height, we essentially rejoin some strings that were already branching. So the same process can take place again with the rejoined strings. Thus, the height of the i-th error tree should behave no worse than i times the height of the trie or the suffix tree. Although this bound may not be very tight, we prove exactly along this intuition. We conjecture that in practice, the heights behave much better.

Theorem 4.3 (Average Data Structure Size and Preprocessing Time). *Let n be the number of strings in the base set S which contains strings and suffixes of strings generated independently and identically distributed at random by a stationary ergodic source satisfying the mixing condition. For any constant d, the average total size of the data structures is $O(n\log^d n)$ and the average time for preprocessing is $O(n\log^{d+1} n)$. Furthermore, these complexities are achieved with high probability $1-o(n^{-\epsilon})$ (for some $\epsilon > 0$).*

Proof. By Lemma 4.2, if there exists an i such that $h_i \geq (2i+1)l$, then we find a string v of length $|v| \geq l$ that is a repeated substring of an independent string or a common substring of two independent strings. Let h^{rep} be the maximal length of any repeated substring in a single independent string in S, let h^{suf} be the maximal length of any repeated substring of a string u for which we have inserted all suffixes into S, and let h^{com} be the maximal length of any common substring of two independent strings. If we bound h^{rep}, h^{suf}, and h^{com} by l, we also bound h_i by $(2i+1)l$. We first turn to long substrings common to independent strings.

For two independent strings r and s, let $C_{i,j} = \max\{k \mid r[i..i+k-1] = s[j..j+k-1]\}$ be the length of a maximal common substring at positions i and j of the two different strings. By the stationarity of the source $\Pr\{C_{i,j} \geq l\} = \Pr\{C_{1,1} \geq l\}$. The latter is the probability that r and s start with the same string of length l, thus $\Pr\{C_{1,1} \geq l\} = \sum_{w\in\Sigma^l}(\Pr\{w\})^2 = \mathbf{E}[w^l]$. For stationary and ergodic sources satisfying the mixing condition, by equation (12), we have $\mathbf{E}[w^l] = \sum_{w\in\Sigma^l}(\Pr\{w\})^2 \to e^{-2r_2 l}$ for $l \to \infty$. By Lemma 4.2, the common substrings must be found in prefixes of length $2il$ of a string in S. As a result, we find

$$\begin{aligned}\Pr\left\{h^{\mathsf{com}} \geq l\right\} \leq \Pr&\left\{\bigcup_{r,s\in S, 1\leq i,j\leq 2il}\{C_{i,j}\geq l\}\right\}\\ &\leq \sum_{r,s\in S,1\leq i,j\leq 2il}\Pr\left\{C_{i,j}\geq l\right\} = \sum_{r,s\in S,1\leq i,j\leq 2il}\mathbf{E}\left[w^l\right]\\ &\qquad\qquad \leq 4n^2l^2i^2\mathbf{E}\left[w^l\right] \leq cn^2l^2i^2e^{-2r_2l}\ ,\end{aligned} \tag{13}$$

for some constant c and growing l.

For the maximal lengths $h^{\textsf{rep}}$ and $h^{\textsf{suf}}$ of repeated substrings, we use known results from [Szp93], where the length $h^{(m)}$ of a repeated substring in a string of length m is bounded by

$$\Pr\left\{h^{(m)} \geq l\right\} \leq c'm\left(l\sqrt{\mathbf{E}\left[w^l\right]} + m\mathbf{E}\left[w^l\right]\right) \leq cmle^{-r_2 l} \tag{14}$$

for some constant c and growing $l > \frac{\ln m}{r_2}$. Because $m \leq 2il$ for $h^{\textsf{rep}}$ and $m \leq |S| = n$ for $h^{\textsf{suf}}$, we can bound

$$\Pr\left\{h^{\textsf{rep}} \geq l\right\} \leq cil^2e^{-r_2 l} \enspace , \tag{15}$$

and

$$\Pr\left\{h^{\textsf{suf}} \geq l\right\} \leq cnle^{-r_2 l} \enspace . \tag{16}$$

Since $h_i \leq (2i+1)\max\{h^{\textsf{rep}}, h^{\textsf{suf}}, h^{\textsf{com}}\}$, we find that

$$\begin{aligned}
\Pr\left\{h_i \geq l\right\} &\leq \Pr\left\{\max\{h^{\textsf{rep}}, h^{\textsf{suf}}, h^{\textsf{com}}\} \geq \frac{l}{2i+1}\right\} \\
&\leq \Pr\left\{h^{\textsf{rep}} \geq \frac{l}{2i+1}\right\} + \Pr\left\{h^{\textsf{suf}} \geq \frac{l}{2i+1}\right\} + \Pr\left\{h^{\textsf{com}} \geq \frac{l}{2i+1}\right\} \\
&\leq c_1\frac{il^2}{(2i+1)^2}e^{-r_2\frac{l}{2i+1}} + c_2n\frac{l}{2i+1}e^{-r_2\frac{l}{2i+1}} + c_3n^2l^2i^2e^{-2r_2\frac{l}{2i+1}} \\
&\leq cn^2l^2i^2e^{-r_2\frac{l}{2i+1}} \enspace ,
\end{aligned} \tag{17}$$

for some constant c and $l > \frac{\ln n}{r_2}$. We condition on $l = (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}$ and get

$$\Pr\left\{h_i \geq (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}\right\} \leq c(1+\epsilon)^2i^4\ln^2 n^{-2\epsilon}n \enspace , \tag{18}$$

for some constant c. Thus, with high probability $1 - o(n^{-\epsilon})$ we have $h_i = O(\log n)$. The expected case can be bounded by

$$\begin{aligned}
\mathbf{E}\left[h_i\right] &\leq (1 - o(n^{-\epsilon}))c\log n + c' \sum_{l \geq (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}} n^2l^3i^2e^{-r_2\frac{l}{2i+1}} \\
&\leq c\log n + c'\sum_{l\geq 0} n^{-2\epsilon}\left(l + (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}\right)^3 i^2e^{-r_2\frac{l}{2i+1}} \\
&\leq c\log n + c''i^5n^{-2\epsilon}\ln^3 n\sum_{l\geq 0} l^3e^{-r_2\frac{l}{2i+1}} \\
&= O\left(\log n\right) \enspace ,
\end{aligned} \tag{19}$$

since $\sum_{l\geq 0} l^3e^{-r_2\frac{l}{2i+1}}$ is convergent. Thus, the expected size of h_i is $O(\log n)$.

This proves the theorem, since $h_i < h_j$ for all $i \leq j$. This suffices to bound the preprocessing time $O(nh_d^{d+1})$ by $O(n\log^{d+1} n)$ and the index size $O(nh_{d-1}^d)$ by $O(n\log^d n)$. □

5 Bounded Preprocessing Time and Space

In the previous section, we achieved a worst-case guarantee for the search time. In this section we describe how to bound the index size in the worst-case in trade-off to having an average-case look-up time. Therefore, we fix h_i in Definitions 3.1 and 3.3 to $h_i = h + i$ for some h to be chosen later but the same for all error trees. By Lemmas 3.4, 3.11, and 4.2, the size and preprocessing time is $O(nh^d)$ and $O(nh^{d+1})$, and the index structure allows to search for patterns w of length $|w| \leq h$ in optimal time $O(|w| + \mathrm{occ})$.

For larger patterns we need an auxiliary structure which is a generalized suffix tree (see, e.g., [Gus97]) for the complete input, i.e., all strings in the base set S. The generalized suffix tree $\mathcal{G}(S)$ is linear in the input size and can be built in linear time. We keep the suffix links that are used in the construction process. For a pattern w, we call a substring v right-maximal, if $v = w[i..j]$ is a substring of some string in S, but $w[i..j+1]$ is not a substring of some string in S. The generalized suffix tree allows us to find all right-maximal substrings of w in time $O(|w|)$. This can be done in the same way as the computation of matching statistics in [CL94]. The approach reminds of the construction of suffix trees: First we compute a canonical reference pair ([Ukk95]) for the largest prefix of w that can be matched in the generalized suffix tree. A canonical reference pair for the substring u is used to represent a virtual node by a node x and a substring v such that $u = \mathrm{path}(x)v$ and $\mathrm{depth}(x)$ is as large as possible. The prefix is right-maximal. Then we take a suffix link from the base of the reference pair replacing the base by the new node. After canonizing the reference pair again, we continue to match characters of w until we find the right-maximal substring starting at the second position in w. This process is continued until the end of w is reached. The total computation takes time $O(|w|)$ because we essentially move two pointers from left to right through w, one for the border of the right-maximal substring and one for the base node of the reference pair. If we build the generalized suffix tree by the algorithm of Ukkonen [Ukk95], this process can also be seen as continuing the algorithm by appending w to the underlying string and storing the lengths of the relevant suffixes.

Assume that t is an i-error length-l match of w. Then, in the relevant edit graph, any path from the start to the end vertex contains i non-zero arcs. But we need at least $|w|$ arcs to get to the end vertex. Thus, there are at least $|w| - i$ zero-weight arcs and there are at least $\frac{|w|-i}{i+1}$ consecutive ones. Thus, there must be a substring of minimal length $\frac{|w|-i}{i+1}$ of w that matches a substring of t exactly.

Since we can handle patterns of length at most h efficiently with our main indexing data structure, we only need to search for patterns of length $|w| > h$ using the generalized suffix tree. For each right-maximal substring v of w, we search at all positions where v occurs in any string in S in a prefix of length at most $|w|$ which we can easily find in the generalized suffix tree using bounded value range queries (see Sections 2.2 and 2.5). For each occurrence we need to search at most $|w|$ positions in time $O(d|w|)$ each. This yields a good algorithm on average if we set $h = c(d+1)\log n$, where n is the cardinality of S, because the probability to find any right-maximal substring of length $c \log n$ is very small.

Lemma 5.1 (Probability of Matching Substrings). *Let w be a pattern generated independently and identically distributed at random by a stationary ergodic source satisfying the mixing condition. Let $|w| = (d+1)l$. The probability that there is a substring u of w of length l that occurs in a prefix of length $|w| + d$ of any string in S is bounded by $cn(|w|+d)|w|e^{-r_{max}l}$ for some constant c.*

Proof. For a stationary ergodic source satisfying the mixing condition, the following limit exists

[Pit85]:

$$r_{max} = \lim_{n\to\infty} -\frac{\max_{t\in\Sigma^n}\{\ln\left(\Pr\left\{t\right\}\right) \mid \Pr\left\{t\right\} > 0\}}{n} \quad . \tag{20}$$

(Observe that r_{max} is positive)

Suppose $S = \{s(1), \ldots, s(n)\}$ and set $C_{i,j,k} = \max\{r \mid s(i)[j..j+r-1] = w[k..k+r-1]\}$. By stationarity of the source, $\Pr\{C_{i,j,k} > l\} = \Pr\{C_{i,j,1} > l\} = \Pr\{s(i)[j..j+l-1]\}$. Since $\Pr\{s(i)[j..j+l-1]\} \leq \max_{t\in\Sigma^l}\Pr\{t\}$, we can apply equation (20) and find that $\Pr\{C_{i,j,k} > l\} \leq ce^{-r_{max}l}$ for some constant c and growing l. As a result we get

$$\begin{aligned} \Pr\left\{|u| > l\right\} = \Pr&\left\{\bigcup_{1\leq k\leq|w|, 0\leq i\leq n, 1\leq j\leq |w|+d} C_{i,j,k} > l\right\} \\ &\leq n(|w|+d)|w|\max_{t\in\Sigma^l}\Pr\left\{t\right\} \leq cn(|w|+d)|w|e^{-r_{max}l} \quad . \end{aligned} \tag{21}$$

□

As a result, for an arbitrary pattern w of length $|w| \geq c'(1+\epsilon)(d+1)\ln n$, we find that the expected work is bounded by

$$cd(|w|)^3(|w|+d)e^{-r_{max}\delta|w|}ne^{-r_{max}c'\ln n} = o(1) \quad , \tag{22}$$

for $\delta = \frac{\epsilon}{1+\epsilon}$ and $c' > \frac{1}{r_{max}}$, while we can find all shorter patterns in optimal time. The size of our data structure is $O(n\log^d n + N)$ and the preprocessing time $O(n\log^{d+1} n + N)$ where N is the size of S.

6 Conclusion and Open Problems

In the context of text indexing, our data structure and search algorithm works best for small patterns of length $O(\log n)$. The average-case analysis shows that these also contribute most. On the other hand, in the worst-case there are more efficient methods for larger strings. The method of Cole et al. [CGL04] starts being useful for large strings of length $\omega(\log n)$ (the d-error neighborhood for strings of length $O(\log n)$ has size $O(\log^d n)$). The method is linear if $m = \Omega(\log^d n)$. For worst-case text indexing, significant progress depends upon the discovery of a linear-time look-up method for medium to large patterns of size $\Omega(\log n) \cap O(\log^d n)$.

Another direction for further research concerns the practical applicability. Although we believe that our approach is fairly easy to implement, we expect the constant factors to be rather large. Therefore, it seems very interesting to study whether the error sets can be thinned out by including less strings. For example, it is not necessary to include errors which “appear” on leaf edges of the preceding error tree. An even more practical question is, whether an efficient implementation without trees based on arrays is possible. First, arrays with all leaves are needed anyway for the range minimum queries. Second, efficient array packing and searching is possible for suffix arrays [AOK02]. For practical purposes a solution for $d \leq 3$ is already desirable.

References

[AKL$^+$00] Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Indexing and dictionary matching with one error. *J. Algorithms*, 37:309–325, 2000.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In A. H. F. Laender and A. L. Oliveira, editors, *Proc. 9th Int. Symp. on String Processing and Information Retrieval (SPIRE)*, volume 2476 of *LNCS*, pages 31–43. Springer, 2002.

[AS92] Alberto Apostolico and Wojciech Szpankowski. Self-alignments in words and their applications. *J. Algorithms*, 13:446–467, 1992.

[BG96] Gerth Stølting Brodal and Leszek Gąsieniec. Approximate dictionary queries. In *Proc. 7th Symp. on Combinatorial Pattern Matching (CPM)*, volume 1075 of *LNCS*, pages 65–74, 1996.

[BGW00] Adam L. Buchsbaum, Michael T. Goodrich, and Jeffery Westbrook. Range searching over tree cross products. In *Proc. 8th European Symp. on Algorithms (ESA)*, volume 1879, pages 120–131, 2000.

[BOR99] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proc. 31st ACM Symp. on Theory of Computing (STOC)*, pages 312–321. ACM Press, 1999.

[BR00] Omer Barkol and Yuval Rabani. Tighter bounds for nearest neighbor search and related problems in the cell probe model. In *Proc. 32nd ACM Symp. on Theory of Computing (STOC)*, pages 388–396. ACM Press, 2000.

[BV00] Gerth Stølting Brodal and Srinivasan Venkatesh. Improved bounds for dictionary look-up with one error. *Information Processing Letters (IPL)*, 75(1–2):57–59, 2000.

[BV02] Paul Beame and Erik Vee. Time-space tradeoffs, multiparty communication complexity, and nearest-neighbor problems. In *Proc. 34th ACM Symp. on Theory of Computing (STOC)*, pages 688–697. ACM Press, 2002.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symp. on Theory of Computing (STOC)*, pages 91–100, 2004.

[CL94] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.

[CN02] Edgar Chávez and Gonzalo Navarro. A metric index for approximate string matching. In *Proc. 5th Latin American Theoretical Informatics (LATIN)*, volume 2286 of *LNCS*, pages 181–195, 2002.

[Cob95] Archie L. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Symp. on Combinatorial Pattern Matching (CPM)*, volume 937 of *LNCS*, pages 41–54. Springer, 1995.

[DLO01] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 289–294. ACM, January 2001.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 137–143, Miami, Florida, USA, October 1997. IEEE.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[GBT84] Harold N. Gabow, Jon Louis Bentely, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th ACM Symp. on Theory of Computing (STOC)*, pages 135–143. ACM, April 1984.

[GMRS03] Alessandra Gabriele, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Indexing structures for approximate string matching. In *Proc. 5th Italian Conference on Algorithms and Complexity (CIAC)*, volume 2653 of *LNCS*, pages 140–151, 2003.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Comp. Science and Computational Biology*. Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffry Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symp. on Theory of Computing (STOC)*, pages 397–406, 2000.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell Systems Technical Journal*, pages 147–160, 1950.

[Ind04] Piotr Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39. CRC Press LLC, 2nd edition, 2004.

[KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 200–210. Springer, 2003.

[KS03] Juha K¨arkk¨ainen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming (ICALP)*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays (extended abstract). In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 186–199. Springer, 2003.

[Lev65] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848, 1965.

[Maa04] Moritz G. Maaß. Average-case analysis of approximate trie search. In *Proc. 15th Symp. on Combinatorial Pattern Matching (CPM)*, volume 3109 of *LNCS*, pages 472–484. Springer, July 2004.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, April 1976.

[MM93] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, October 1993.

[MN05] Moritz G. Maaß and Johannes Nowak. A new method for approximate indexing and dictionary lookup with one error. *Information Processing Letters (IPL)*, To be published 2005.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*. ACM/SIAM, 2002.

[Mye94] Eugene W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

[NBY00] Gonzalo Navarro and Ricardo Baeza-Yates. A hybrid indexing method for approximate string matching. *J. Discrete Algorithms*, 1(1):205–209, 2000. Special issue on Matching Patterns.

[NBYST01] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, December 2001.

[Now04] Johannes Nowak. A new indexing method for approximate pattern matching with one mismatch. Master's thesis, Fakult¨at f¨ur Informatik, Technische Universit¨at M¨unchen, Boltzmannstr. 3, D-85748 Garching, February 2004.

[Pit85] Boris Pittel. Asymptotical growth of a class of random trees. *Annals of Probability*, 13(2):414–427, 1985.

[Szp93] Wojciech Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transact. on Information Theory*, 39(5):1647–1659, September 1993.

[Szp00] Wojciech Szpankowski. *Average Case Analysis of Algorithms on Sequences*. Wiley-Interscience, 1st edition, 2000.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ukk93] Esko Ukkonen. Approximate string-matching over suffix trees. In *Proc. 4th Symp. on Combinatorial Pattern Matching (CPM)*, volume 684 of *LNCS*, pages 228–242. Springer, 1993.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[YY95] Andrew C. Yao and Frances F. Yao. Dictionary look-up with small errors. In *Proc. 6th Symp. on Combinatorial Pattern Matching (CPM)*, volume 937 of *LNCS*, pages 387–394, 1995.

[YY97] Andrew C. Yao and Frances F. Yao. Dictionary look-up with one error. *J. Algorithms*, 25:194–202, 1997. journal of YaoYao95.