Text Indexing with Errors

Moritz G. Maaß* and Johannes Nowak

Fakultät für Informatik, Technische Universität München
Boltzmannstraße 3, D-85748 Garching, Germany
{maass,nowakj}@informatik.tu-muenchen.de

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 lookup time is not linear or depends upon the size of the document corpus. Our data structure has size $O(n \log^k n)$ on average and with high probability for input size n and queries with up to k 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 lookup 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 online queries vital. On the other hand, the nature of the data frequently calls for error-tolerant methods: 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, matching with errors is useful even for immaculate data, e.g., for similarity searches. For online 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 [22]). Recently, some progress has been made towards the construction of error-tolerant text indexes [2, 6, 10], but in general the problem remains open.

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

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

A. Apostolico, M. Crochemore, and K. Park (Eds.): CPM 2005, LNCS 3537, pp. 21–32, 2005.

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 occurrences of a pattern. The natural lower bound, linear $\Theta(n)$ size (preprocessing time) and linear $\Theta(m + \text{occ})$ lookup time, can be reached[1] for exact matching (e.g., [19, 31, 32]). For edit or Hamming distance, no index with lookup time $O(m + \text{occ})$ 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 lookup time depends on n or is not linear in m.

2 Our Results

There appear various definitions of approximate text indexing in the literature. The broader definition just requires the index to "speed up searches" [24], while a stricter approach requires to answer on-line queries "in time proportional to the pattern length and the number of occurrences" [2]. Our algorithm satisfies even the latter definition. We present and analyze a new index structure for approximate pattern matching problems allowing a constant number of $k = O(1)$ errors. The method works for various problem flavors, e.g., approximate text indexing (full text indexing), approximate dictionary lookup indexing (word based indexing), and approximate document collection indexing (see below). For all of these problems we achieve an optimal worst-case lookup time of $O(m + \text{occ})$ employing an index that uses $O(n \log^k n)$ additional space and requires preprocessing time $O(n \log^{k+1} n)$, both on average and with high probability. For approximate dictionary lookup these bounds even improve to $O(|C| \log^k |C|)$ additional space and $O(|C| \log^{k+1} |C|)$ 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. For the efficient retrieval of results, range queries are prepared on the leaves of the trees. 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., [28]). To our knowledge, this yields the first reasonable sized indexes achieving optimal lookup time. Additionally, we present a trade-off between query time and index complexity, achieving worst-case bounded index size $O(n \log^k n)$ and preprocessing time $O(n \log^{k+1} n)$ while having linear lookup time $O(m + \text{occ})$ on average.

3 Related Work

A survey on text indexing is given by Navarro et al. [24]. For the related nearest neighbor problem see the survey by Indyk [15]. For approximate dictionary indexing with a set of n strings of length m, one can construct a data structure that supports queries in $O(m)$ and has size $O(nm)$ [5]. For text indexing (on a single text of length n) we summarize the results in Table 1.

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

Table 1. Previous work on text indexing. (Results marked "avg" are achieved on average and results marked "whp" are achieved with high probability.)

Errors	Model	Query Time	Index Size and Preparation Time (if different)	Literature
none	exact	$O(m+\text{occ})$	$O(n)$	Weiner 1973 [32]
1	edit	$O(m \log n \log\log n+\text{occ})$	$O(n \log^2 n)$	Amir et al. 2000[2]
1	edit	$O(m \log\log n+\text{occ})$	$O(n \log n)$	Buchsbaum et al. 2000 [6]
1	Ham.	$O(m+\text{occ})$	$O(n \log n)$, $O(n \log^2 n)$ (avg,whp)	N 2004 [25]
1	edit	$O(m+\text{occ})$	$O(n \log n)$, $O(n \log^2 n)$ (avg,whp)	MN 2004 [18]
$O(1)$	edit	$O(m \min\{n, m^{k+1}\}+\text{occ})$	$O(\min\{n, m^{k+1}\}+n)$	Cobbs 1995 (Ukkonen 1993) [9, 30]
$O(1)$	edit	$O(kn^{\epsilon} \log n)$	$O(n)$	Myers 1994 [21]
$O(1)$	edit	$O(n^{\epsilon})$	$O(n)$	Navarro et al. 2000 [23]
$O(1)$	edit	$O(m \log^2 n+m^2+\text{occ})$ (avg)	$O(n \log n)$, $O(n \log^2 n)$ (avg)	Chávez et al. 2002 [8]
$O(1)$	edit	$O(m+\log^k n \log\log n+\text{occ})$	$O(n \log^k n)$	Cole et al. 2004 [10]
$O(1)$	Ham.	$O(\log^{k+1} n)$ (avg)	$O(n)$, $O(n)$	M 2004 [16]
$O(1)$	edit	$O(m+\text{occ})$	$O(n \log^k n)$, $O(n \log^{k+1} n)$ (avg,whp)	This paper
$O(1)$	edit	$O(m+\text{occ})$ (avg,whp)	$O(n \log^k n)$, $O(n \log^{k+1} n)$	This paper
k mismatches in a window of length r		$O(m+\text{occ})$ (avg)	$O(n \log^l n)$ (avg)	Gabriele et. al 2003 [13]

Due to the limited space, the most proofs are omitted. The interested reader is referred to [17] for more details and proofs.

4 Preliminaries

Let Σ be any finite alphabet and let $|\Sigma|$ denote its size. We consider $|\Sigma|$ to be constant. The set of all strings including the empty string ε is denoted by Σ^*. Let $t = t[1] \cdots t[n] \in \Sigma^n$ be a string of length $|t| = n$. 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 $t[i..j] = t[i]t[i+1] \cdots t[j]$, $\text{pref}_k(t) = t[1..k]$, and $\text{suff}_k(t) = t[k..n]$ (with $t[i..j] = \varepsilon$ for $i > j$). For $u, v \in \Sigma^*$ we let $u \sqsubseteq_{\mathsf{pref}} v$ denote that $u = \text{pref}_{|u|}(v)$. 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 $\text{maxpref}(S) = |u|$. The size of S is defined as $\sum_{u \in S} |u|$.

We use the well-known *edit distance* (Levenshtein distance) to measure distances between strings. The edit distance of two strings u and v, $\text{d}(u, v)$, 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 $O(|u||v|)$. It can also be defined via the operators del, ins, and sub of type $\Sigma^* \to \Sigma^*$. If for two strings $u, v \in \Sigma^*$ we have distance $\text{d}(u, v) = k$, then there exist one or more sequences of operations $(op_1, op_2, \ldots, op_k)$, $op_i \in \{\text{del}, \text{ins}, \text{sub}\}$, such that $v = op_k(op_{k-1}(\cdots op_1(u) \cdots))$. 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. In this light, *Hamming distance* is only a simplified version of edit distance, so we focus our attention on edit distance (Hamming distance is simpler and might help the intuition, especially for the next definition). The minimal prefix of a string u that contains all errors with respect to comparison to a string v plays an important role. For two strings $u, v \in \Sigma^*$ with $\mathrm{d}(u, v) = k$, we define the k-minimal prefix length $\mathrm{minpref}_{k,u}(v)$ as the minimal value l for which $\mathrm{d}(\mathrm{pref}_l(u), \mathrm{pref}_{l+|v|-|u|}(v)) = k$ and $\mathrm{suff}_{l+1}(u) = \mathrm{suff}_{l+1+|v|-|u|}(v)$. In other words, all errors between u and v are located in a prefix of length $\mathrm{minpref}_{k,u}(v)$ of u.

For fast lookup, strings can be stored in a *trie*. A trie $\mathcal{T}(S)$ 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 $|\mathrm{path}(x)|$. 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 subtrie rooted at x. For convenience we also denote $\mathcal{T}_u = \mathcal{T}_{\mathrm{path}(x)} = \mathcal{T}_x$. The string height of $\mathcal{T}$ is defined as $\mathrm{height}(\mathcal{T}) = \max\{|\mathrm{path}(x)| \mid x \text{ is an inner node of } \mathcal{T}\}$ which 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 (compare the representation in [31]). Thus, all previous definitions for tries apply to compact tries as well, possibly using virtual instead of existing nodes. We only use compact tries, and we denote the compact trie for the string set S by $\mathcal{T}(S)$.

If a (compact) trie is never searched deeper than string depth l, i.e., below string depth l we only need to enumerate leaves, then we can drop the unique branching criterion on outgoing edges below this level. Weak tries are needed to bound the preprocessing time and space in the worst case in Section 7.

Definition 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 $\mathcal{W}_{\mathrm{maxpref}(S)}(S) = \mathcal{T}(S)$.

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. The following range queries all operate on arrays

containing integers. The goal is to preprocess these arrays in linear time so that the queries can be answered in constant time. Assume that A of size n is an array containing integer values (indexed 1 through n). A *bounded value range query* (BVR) for the range (i, j) with bound k on array A asks for the set of indices L in the given range where A contains values less than or equal to k. The set is given by $L = \{l \mid i \leq l \leq j \text{ and } A[l] \leq k\}$. A *colored range query* (CR) for the range (i, j) on array A asks for the set of distinct elements in the given range of A. These are given by $C = \{A[k] \mid i \leq k \leq j\}$. BVR queries can be answered with $O(n)$ preprocessing time and space and $O(|L|)$ query time [20], based on the well-known range minimum query (RMQ) problem which can be solved with $O(n)$ preprocessing and $O(1)$ query time [12]. CR queries can also be solved with $O(n)$ preprocessing time and space and $O(|C|)$ query time [20].

There are many different *text indexing* problems. 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. When matching a pattern allowing k errors against the text database, we can report *occurrences* (i.e., the exact starting and ending points of a matching substring), *positions* (only the starting points), or – if we have more than one text – *documents* (in which the pattern can be matched). The number of outputs decreases from occurrence to position to document reporting. To ease further exposition, we take a unified view by considering our text database to be a set S of n strings, called the *base set*. The indexing problems handled by our approach are named by the structure of strings in the base set S. If S is the set of suffixes of a string T, we have the k-Approximate Text Indexing (k-ATI) problem. If S is a collection of independent strings, we have the k-Approximate Dictionary Indexing (k-ADI) problem. Finally, if S is the set of suffixes of a collection of documents, we have the k-Approximate Dictionary Indexing (k-ADI) problem. Note that in addition to the three types of reported outputs, for the (k-ADI) problem, it also makes sense to report only documents completely matched by the pattern, called *full hits*. Observe that in all instances in addition to the size of the (originally) given strings a (compact) trie for the underlying collection consumes $O(n)$ space by using pointers into strings instead of substrings.

5 The Basic Index Data Structure

In each of our indexing problems, in order to answer a query we have to find prefixes of strings in the base set S that match the search pattern P with k errors. We call $s \in S$ a *k-error, length-l occurrence of P* if $\mathrm{d}(\mathrm{pref}_l(s), P) = k$. We then also say that P "matches" s up to length l with k errors. To solve the various indexing problems defined in the previous section, we need to be able to find all k-error occurrences of P efficiently.

The basic idea of our index for k errors is to make $k + 1$ case distinctions on the position of the first i errors between the pattern P and a k-error match. If P matches a prefix $t \sqsubseteq_{\mathsf{pref}} s$ of a string $s \in S$, we first look whether there is an

exact match of a prefix of t and P that is larger than the height h_0 of the trie $\mathcal{T}$ for S. There can be at most one such string, and we check it in time $O(|P|)$ (we can check for an i-error match in time $O(i|P|)$, see, e.g., [29]). Otherwise, there is at least one error before h_0. We build a trie $\mathcal{T}'$ of all strings that contain one error before h_0 and search again for a prefix of P that extends above the height h_1 of $\mathcal{T}'$. If it exists, we check the corresponding string in time $O(|P|)$. Otherwise, there is at least another error before h_1, and we build the next trie $\mathcal{T}''$ by including the errors before h_1. We continue this case distinction until we have either reached the k-th trie or we can match the complete pattern. In the latter case, all leaves in the subtree below P are matches if all errors with the original string are in a prefix of the length of the pattern. We select the appropriate leaves using range queries on the i-minimal prefix lengths. Thus, we either check a single string in the i-th trie in time $O(i|P|)$, or we select a number of matches with range queries in time $O(|P| + \mathrm{occ})$.

More generally, we use the parameters $h_0, \ldots, h_k$ to control the precomputing. The key property of the following inductively defined sets $W_0, \ldots, W_k$ is that we have at least one error before $h_0 + 1$, two errors before $h_1 + 1$, and so on until we have k errors before $h_{k-1} + 1$ in W_k. The partitioning into $k+1$ sets allows searching with different error bounds up to k and enables the efficient selection of occurrences.

Definition 2 (Error Sets). *For $0 \leq i \leq k$, the error set W_i are defined inductively. The first string set W_0 is defined by $W_0 = S$. The other sets are*

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

where the operator Γ_l is defined by $\Gamma_l(A) = \{op(u)v \mid uv \in A, |op(u)| \leq l + 1, op \in \{\mathrm{del}, \mathrm{ins}, \mathrm{sub}\}\}$ on sets of strings $A \subset \Sigma^*$, S_i is defined as $S_i = \{r \mid$ there exists $s \in S$ such that $\mathrm{d}(s, r) = i\}$ (i.e., the strings within distance i of S), and h_i are integer parameters (that may depend on W_i).

The set W_i may consist of independent strings and of suffixes of independent strings. We assume that independent strings do not match prefixes of each other with $2k$ or less errors, i.e., for any $s \neq t \in S$ and any prefixes $u \sqsubseteq_{\mathsf{pref}} s$ we have $\mathrm{d}(u, t) > 2k$. This can be achieved by adding sentinels at the end of each string. Suffixes of the same strings can match each other, but there can be at most $2k + 1$ suffixes from the same string that match any string of length l.

Lemma 1 (String Set Sizes). *The size of the string sets W_i is bounded by $|W_i| = O(h_{i-1}|W_{i-1}|)$.*

To search the error sets efficiently, while not spending too much time in preprocessing, we use weak tries, which we call *error trees*.

Definition 3 (Error Trees). *The i-th error tree $\mathrm{et}_i(S)$ is defined as $\mathrm{et}_i(S) = \mathcal{W}_{h_i}(W_i)$. Each leaf x in $\mathrm{et}_i(S)$ is labeled (id_s, l) if id_s is an identifier for $s \in S$, $\mathrm{d}(s, \mathrm{path}(x)) = i$, and $l = \mathrm{minpref}_{i,\mathrm{path}(x)}(s)$.*

The correctness of our approach derives from the proof of the following three key properties (assuming either $h_i > \mathrm{maxpref}(W_i)$ or $P \leq h_i$):

1. Let t be a prefix of $s \in S$ and assume $l = \text{minpref}_{i,t}(P)$. If
   (i) t matches P with i errors, i.e., $\text{d}(t, P) = i$, and
   (ii) there exists an ordered edit sequence such that $\text{minpref}_{j,t_j}(t) \leq h_{i-1} + 1$, where for all $0 \leq j \leq i$, $t_j = op_j(\cdots op_1(t)\cdots)$ are the edit stages,

   then there is a leaf x labeled (id_s, l) in $\text{et}_i(S)_P$. In other words, if the first i edit operations occur before the bounds $h_j + 1$, then the i-error occurrence is covered by precomputing.
2. Let x be a leaf in $\text{et}_i(S)_P$ with label (id_s, l) corresponding to a string $s \in S$. There exists $t \sqsubseteq_{\text{pref}} s$ with $\text{d}(t, P) = i$ if and only if $l \leq |P|$. Thus, not all leaves found in $\text{et}_i(S)_P$ correspond to i-error occurrences of P. To locate the correct leaves, we use range queries, see Section 6.
3. Finally, if P matches a prefix t of some string $s \in S$ with exactly i errors, then there is a dichotomy. Let $\rho(P, t) = (op_1, op_2, \ldots, op_i)$ be an ordered edit sequence.

   Case A Either P can be matched completely in the i-th error tree $\text{et}_i(S)$ and a leaf x labeled (id_s, l) can be found in $\text{et}_i(S)_P$, or

   Case B there exists an edit operation in $\rho(P, t)$ that occurs after some h_j, i.e., a prefix $p \sqsubseteq_{\text{pref}} P$ of length $|p| > h_j$ is found in $\text{et}_j(S)$ and $\text{et}_j(S)_p$ contains a leaf x with label (id_s, l).

To capture the intuition first, it is easier to assume that we set $h_i = \text{maxpref}(W_i)$, where the error trees become compact tries. Note that a leaf can have at most $2k+1$ labels. The search algorithms are applications of these key properties. The basic search data structure consists of the $k + 1$ error trees $\text{et}_0(S), \ldots, \text{et}_k(S)$. The elementary step is to match P in each of the $k + 1$ error trees $\text{et}_i(S)$ for $0 \leq i \leq k$. Property 3 gives two relevant cases that must be handled.

6 Text Indexing with Worst-Case Optimal Search-Time

To achieve worst-case optimal search-time, we set h_i in Definitions 2 and 3 to the string height of the error tree $h_i = \text{maxpref}(W_i)$. It immediately follows that Case B does not contribute to the main search complexity, since each leaf represents at most $2k + 1$ different strings from the base set S. We have $k + 1$ error trees, thus, for constant k, the total search time spent for Case B is $O(m)$.

The more difficult part is Case A. Some of the leaves in the subtree $\text{et}_i(S)_P$ may not correspond to i-error matches (i.e., not satisfying the second property above), we cannot afford to traverse the complete subtree. Let n_i be the number of leaf labels in $\text{et}_i(S)$. We create an array A_i of size n_i containing pointers to the leaves of $\text{et}_i(S)$. At each inner node x we store pointers $\text{left}(x)$ and $\text{right}(x)$ to the left-most and right-most index in A_i of the leaves in the subtree rooted at x. Depending on the desired output, we create additional arrays B_i from the leaf labels of each error tree $\text{et}_i(S)$ and prepare them for CR or BVR queries to select a desired subset in output optimal time. This takes time and space $O(n_i)$.

Theorem 1 (Worst-Case Search Times). *Let m be the length of the pattern P and* occ *be the number of outputs. Using $k+1$ error trees and additional data structures linear in the size of the error trees, we can solve the problems k-ATI, k-ADI, k-ADCI with a worst-case lookup time of $O(m + \text{occ})$.*

Proof (Sketch). Case B takes time $O(m)$. Observe that for leaf label (id_s, l) in $\text{et}_i(S)_P$, a prefix of s matches P with at most i errors. Depending on the type of indexing problem we create the arrays B_i and prepare range queries on the array. For each indexing problem, we have different string identifiers, e.g., for k-ADCI id_s is a combination of the string number and the suffix number of the string. We use this and the annotated k-minimal prefix length l to prepare the range queries. For example, to report occurrences, we set $B_i[j] = l$ if the leaf stored in $A[j]$ has label (id_s, l). From the root x of $\text{et}_i(S)_P$ we then perform a BVRQ $(\text{left}(x), \text{right}(x), |P|)$ on B_i, which yields exactly the indexes of the leaves representing occurrences. Note that the structure of $\text{et}_i(S)_P$ (whether it is a trie or not) has no effect on the complexity. The pattern P is searched k times, and range queries are linear in their output, so the running time is $O(m + \text{occ})$. □

Compact (weak) tries with n leaves can be implemented in size $O(n)$ plus the size of the underlying strings. The index size is bounded in the following theorem.

Theorem 2 (Data Structure Size and Preprocessing Time). *Let n be the number of strings in S, and let $h = \max_{0 \le i \le k} h_i$. Then, for constant k, the total size of the data structures is $O(nh^k)$ and the preprocessing time is $O(nh^{k+1})$.*

Proof (Sketch). By Lemma 1, the size of the k-th error tree dominates the total size. The total number of leaves of all error trees is $O(nh_0 \cdots h_{k-1})$, which also bounds the size of the range query data structures and time to build them. Each error tree is built naively, copying with errors from the preceding tree. To eliminate duplicates, strings are merged up to $h + k$ so that no errors occur in the remaining suffixes. Duplicates can than be eliminated using $O(|S|k)$ buckets to sort the strings with a common prefix of length $h + k$ in linear time by their string identifier in S and the suffix length in $\{h, \ldots, h + 2k\}$. In the error tree, strings are merged up to depth h_k only. □

Note that in the worst-case $h_i = \Omega(n)$. The average-case is fortunately much better for a wide range of probabilistic models, and it occurs with high probability. The maximal string depth of any error tree is $O(\log n)$ on average and with high probability for a wide range of probabilistic models. For the analysis, we assume that all strings are generated independently and identically distributed at random by a stationary ergodic source satisfying the *mixing* condition (see, e.g., [28]). Let $\mathcal{F}_n^m$ be the σ-field generated by $\{X_k\}_{k=n}^m$ for $n < m$, then $\{X_k\}_{k=-\infty}^{\infty}$ satisfies the mixing condition, if there exist constants $c_1, c_2 \in \mathbb{R}$ and $d \in \mathbb{N}$ such that for all $-\infty \le m \le m + d \le n$ and for all $\mathcal{A} \in \mathcal{F}_{-\infty}^m$, $\mathcal{B} \in \mathcal{F}_{m+d}^{\infty}$, we have $c_1 \Pr\{\mathcal{A}\}\Pr\{\mathcal{B}\} \le \Pr\{\mathcal{A} \cap \mathcal{B}\} \le c_2 \Pr\{\mathcal{A}\}\Pr\{\mathcal{B}\}$. Note that this model includes independent trials and stationary ergodic Markov chains. In this model, one can prove that the probability that there is a repeated substring of length $(1+\epsilon)\frac{\ln n}{r_2}$

in any string of length n is bounded by $cn^{-\epsilon} \ln n$ (where c and r_2 are constants with r_2 depending on the probability distribution) [27]. Using results from [26], it is possible to prove that the probability that there is a common substring of length $(1+\epsilon)\frac{l}{r_2}$ between the prefixes of length l' of any two strings in a set of n independent strings is bounded by $cn^2 l'^2 e^{-2(1+\epsilon)l}$. Thus, the probability of a repeated substring of length $(1+\epsilon)\frac{\ln n}{r_2}$ in a prefix of length $c'(1+\epsilon)\ln n$ is bounded by $c'n^{-2\epsilon}\ln^2 n$.

If the height of the k-th error tree is h, then there are two prefixes of strings in S within distance $2k$. Thus, there exists a common substring between this strings of length $\Omega(\frac{h}{2k})$. Therefore, if the strings are independent, we find that for constant k the expected height cannot exceed $O(\log n)$. To handle the case that both prefixes belong to suffixes of the same string, we have to prove that there is actually a repeated substring of length $\Omega(\frac{h}{2k})$. The idea for this proof is as follows: We assume that the height of the i-th error tree is larger than $ci \log n$ and prove that either there is such a substring, or that the size of the height of the $(i-1)$-th error tree is larger than $c(i-1)\log n$. Note that completely identical strings are joined since W_i are sets. Together this yields the following theorem.

Theorem 3 (Average Data Structure Size and Preprocessing Time). *Let n be the number of strings in S. Then, for any constant k, the average total size of the data structures used is $O(n \log^k n)$ and the average time for preprocessing is $O(n \log^{k+1} n)$. Furthermore, these complexities are achieved with high probability $1 - o(n^{-\epsilon})$ (for $\epsilon > 0$).*

7 Bounded Preprocessing 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 lookup time. Therefore, we fix h_i in Definitions 2 and 3 to $h_i = h + 1 = c \log n + i$. By Theorem 2, the size of all trees is then $O(n \log^k n)$. For patterns P of length $|P| < h$, we use the same data structures and algorithms as described in the previous Section. Note that Case B never occurs. Since $|P| < h$, all errors must have occurred before h in the respective error tree and are thus accounted for. This yields the following corollary.

Corollary 1 (Small Patterns). *Let P be a pattern of length $m < c \log n$ for some constant c, and let* occ *be the number of outputs. The problems k-ATI, k-ADI, and k-ADCI can be solved with an index of size $O(n \log^k n)$ which is built in time $O(n \log^{k+1} n)$ and with lookup time $O(m + \text{occ})$.*

For larger patterns we need an auxiliary structure, which is a generalized suffix tree (see, e.g., [14]) for the complete input, i.e., all strings in S. The generalized suffix tree $\mathcal{G}(S)$ is linear in the input size and can be built in linear time (e.g., [19, 31]). We keep the suffix links that are used in the construction process. A suffix link is an auxiliary edge that points from a node x with $\text{path}(x) = av$ to a node y with $\text{path}(y) = v$, thus, allowing to "chop off" characters at the

front of the current search string. For any pattern P of length m this allows to find all nodes x such that path(x) represents a right-maximal substring $P[i..j]$ of P in time $O(m)$ (see, e.g., the computation of matching statistics in [7]). The pattern P can match a string $s \in_{\mathsf{pref}} S$ with k errors only if a substring $P[i..j]$ of length $\frac{m}{k+1} - 1$ matches a substring of s exactly. Thus, we find all positions of such substrings and check in time ckm^2 whether there is an occurrence within m characters of each position. Assuming that P is generated by an ergodic stationary source satisfying the mixing condition, we can prove that there are only very few positions and get the following theorem.

Theorem 4 (Large Patterns). *Let P be a pattern of length m and let* occ *be the number of outputs. Let n be the cardinality of S. The problems k-ATI, k-ADI, and k-ADCI can be solved with an index of size $O(n \log^k n)$ which is built in time $O(n \log^{k+1} n)$ and with lookup time $O(m + \mathrm{occ})$ on average and with high probability.*

Proof (Sketch). We set h to $c(k+1)\log n + 2k$ and handle all small pattern by Corollary 1. The probability that a substring of P of length at least l is found can be bounded by $nm^2 e^{-r_{max} l}$ (where r_{max} is a constant depending on the probability distribution [26]). Therefore, the probability to to perform any work is bounded by $m^2 n^{1-r_{max}(c)} = n^{-\epsilon}$ for suitably chosen c. Thus, the total expected work for large patterns is $O(1)$. □

8 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. [10] starts being useful for large strings of length $\omega(\log n)$ (the k-error neighborhood for strings of length $O(\log n)$ has size $O(\log^k n)$). The method is linear if $m = \Omega(\log^k n)$. For worst-case text indexing, significant progress depends upon the discovery of a linear-time-lookup method for medium to large patterns of size $\Omega(\log n) \cap O(\log^k n)$.

Our work together with [10] and [16] seems to indicate that – compared to exact searching – an additional complexity factor of $\log^k n$ is inherent. Regarding the index space, a linear lower bound has been proven for the exact indexing problem [11]. However, for the approximate indexing problem, lower bounds do not seem easy to achieve. Using asymmetric communication complexity some bounds for nearest neighbor search in the Hamming cube can be shown [3, 4], 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 information theoretic method of [11] also seems to fall short because approximate lookup does not improve compression and there is no restriction on the lookup time.

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. Arrays with all leaves are needed anyway for the range minimum queries. Secondly, efficient array packing and searching is possible for suffix arrays [1]. For practical purposes a space efficient solution for $k \leq 3$ is already desirable.

Acknowledgements

We thank the anonymous referees for their insightful comments.

References

1. M. I. Abouelhoda, E. Ohlebusch, and S. 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.
2. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. *J. Algorithms*, 37:309–325, 2000.
3. O. Barkol and Y. 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.
4. A. Borodin, R. Ostrovsky, and Y. 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.
5. G. S. Brodal and L. Gąsieniec. Approximate dictionary queries. In *Proc. 7th Symp. on Combinatorial Pattern Matching (CPM)*, volume 1075 of *LNCS*, pages 65–74, 1996.
6. A. L. Buchsbaum, M. T. Goodrich, and J. Westbrook. Range searching over tree cross products. In *Proc. 8th European Symp. on Algorithms (ESA)*, volume 1879, pages 120–131, 2000.
7. W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.
8. E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. 5th Latin American Theoretical Informatics (LATIN)*, volume 2286 of *LNCS*, pages 181–195, 2002.
9. A. 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.
10. R. Cole, L.-A. Gottlieb, and M. 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.
11. E. D. Demaine and A. 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, Jan. 2001.

12. H. N. Gabow, J. L. Bentely, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th ACM Symp. on Theory of Computing (STOC)*, pages 135–143. ACM, Apr. 1984.
13. A. Gabriele, F. Mignosi, A. Restivo, and M. 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.
14. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Comp. Science and Computational Biology*. Cambridge University Press, 1997.
15. P. Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39. CRC Press LLC, 2nd edition, 2004.
16. M. 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.
17. M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-I0503, Fakultät für Informatik, TU München, Mar. 2005.
18. M. G. Maaß and J. Nowak. A new method for approximate indexing and dictionary lookup with one error. *Information Processing Letters (IPL)*, To be published.
19. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.
20. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*. ACM/SIAM, 2002.
21. E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.
22. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, Mar. 2001.
23. G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. Discrete Algorithms*, 1(1):205–209, 2000. Special issue on Matching Patterns.
24. G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, Dec. 2001.
25. J. Nowak. A new indexing method for approximate pattern matching with one mismatch. Master's thesis, Fakultät für Informatik, Technische Universität München, Boltzmannstr. 3, D-85748 Garching, Feb. 2004.
26. B. Pittel. Asymptotical growth of a class of random trees. *Annals of Probability*, 13(2):414–427, 1985.
27. W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transact. on Information Theory*, 39(5):1647–1659, Sept. 1993.
28. W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. Wiley-Interscience, 1st edition, 2000.
29. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
30. E. 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.
31. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
32. P. Weiner. Linear pattern matching. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11. IEEE, 1973.