LUDWIG-MAXIMILIANS-UNIVERSITÄT
TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik; Lehrstuhl für Effiziente Algorithmen

Diplomarbeit

A New Indexing Method For Approximate Pattern Matching With One Mismatch

Johannes Nowak

Aufgabensteller: Prof. Dr. Ernst W. Mayr
Betreuer: Moritz G. Maaß
Abgabedatum: 15.02.2004

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

15.02.2004 ______________________

Johannes Nowak

Abstract

We consider the problem of designing an efficient index for approximate pattern matching. Despite ongoing efforts this topic is still a challenge in combinatorial pattern matching.

We present a new index structure that allows to report all substrings of a text that match a given input pattern with one mismatch in linear time with respect to the pattern length. The size of our index is $O(n \log n)$ on average and with high probability, where n is the length of the text. The index structure is a Σ^+-tree which originates from the suffix tree and the merging of certain subtrees of it. Our method is conceptually very easy and intuitive.

To achieve these results we use suffix trees and constant time lowest common ancestor queries. The time and space complexity of the index structure depends on the height of the corresponding suffix tree, which is proven to be $O(n \log n)$ with high probability for a variety of probabilistic models.

Zusammenfassung

Wir beschäftigen uns in dieser Arbeit mit der Entwicklung einer effizienten Indexstruktur für Pattern Matching mit Fehlern. Obwohl auf diesem Gebiet in den letzten Jahren Fortschritte erziehlt wurden, gibt es noch immer keine Lösung für dieses Problem, die mit der Effizienz von Methoden für exaktes Pattern Matching vergleichbar wäre.

Wir präsentieren eine neue Indexstruktur, mit deren Hilfe man alle Teilstrings eines Textes, die mit einem gegebenes Eingabewort mit höchstens einer Substitution übereinstimmen, in linearer Zeit, bezüglich der Länge des Eingabeworts, finden kann. Der Speicherbedarf der Struktur ist $O(n \log n)$ im Average Case und zudem mit hoher Wahrscheinlichkeit, wobei n die Länge des Textes ist. Die Methode ist konzeptionell sehr einfach und intuitiv.

Bekannte Datenstrukturen und Algorithmen, die in unserem Ansatz verwendet werden, sind hauptsächlich Suffixbäume und die Bestimmung des niedrigsten gemeinsamen Vorfahrens zweier Knoten in einem Baum in konstanter Zeit. Die Zeit- und Speicherkomplexität unserer Indexstruktur ist in erster Linie von der Höhe des zugrunde liegenden Suffixbaumes abhängig. Es wurde für eine Vielzahl von wahrscheinlichkeitstheoretischen Modellen bewiesen, dass dieser Parameter mit hoher Wahrscheinlichkeit $O(\log n)$ ist.

Contents

Chapter 1

Introduction

Approximate pattern matching is a problem that arises in many areas of computer science including text searching and computational biology. In its most general form approximate pattern matching is about finding occurrences of a *pattern* in a *text* allowing a given number or percentage of errors. This work focuses on the problem of building efficient indices on the text, that is constructing a persistent data structure on the text in a preprocessing step that improves later searches of patterns allowing errors. We start by describing the problem in greater detail and motivating it by a few applications.

1.1 The Problem

There are two variants of the approximate pattern matching problem. The first is the so-called online searching, where the pattern can be preprocessed but not the text. The second is indexed searching. Here, in a preprocessing step a persistent data structure is built that is used to speed up later searches. Most commonly, such an index is built in the case that the text is large and on the assumption that there will be frequent queries with different, but considerably shorter patterns. Some applications, however, require the matching of some given patterns against several texts. This problem is also referred to as the *dictionary matching problem*, and in that case the index is built on a set of patterns [MP68]. In any case, the time needed for preprocessing and the space needed for storing the data structure have to amortize over many searches in order to justify the expenses.

In this work we focus on the problem of building an index structure on a given input text in order to speed up approximate matching of various input patterns. This problem is of great importance and an area of ongoing scientific research. Many of such indexing methods have been developed for exact pattern matching [AG97, Gus97], providing both time and space efficient solutions. If we take the efficiency of those indexes as a benchmark, nothing comparable has been found for the approximate pattern matching problem so far. The authors of [NBYST01] conclude by saying, "Radical innovative ideas are welcome on this area," and thus further research and the development of new methods are still necessary.

Good surveys on approximate pattern matching are available. We refer to Navarro [Nav01], which offers a good overview of the most common methods for solving the online version of the problem, and Navarro et al. [NBYST01] for the state of the art of the indexing problem.

The contribution of this work is a new indexing structure that allows finding all substrings of a text that match an arbitrary input pattern with one mismatch in linear time with respect to the pattern size in the worst case. To our knowledge, this structure is the first that achieves that efficiency with having an average space complexity of $O(n \log n)$. Moreover, the size of our index structure is $O(n \log n)$ with high probability. This work is organized as follows. In Chapter 2 we give basic definitions of the data structures and algorithms that are commonly known, but also important to our method – above all *suffix trees* and the constant time solution to the *lowest common ancestor problem*. In Chapter 3 we introduce our new index structure along with a thorough analysis of its complexity. The main results of this analysis are captured in Theorems 3.11, 3.52, and 3.53.

We continue motivating the problem with a few applications, mainly from computational biology.

1.2 Motivating Examples

The main motivation for approximate pattern matching was given by computational biology, signal processing, and text retrieval problems and dates back to the sixties and seventies [Nav98].

Nucleic acids and proteins are the central macromolecules in the biochemical activity of all livings. These macromolecules can be considered as long sequences or texts over specific alphabets. In the case of proteins the alphabet is a set of twenty amino acids, whereas nucleic acids (DNA or RNA) are composed of four different nucleotides or bases. Although this consideration abstracts away the three-dimensional, dynamic nature of these molecules, a lot of biologically important problems can be considered on the level of sequences [Les94, Gus97]. The number of these problems is great, and we refer to books about computational biology to get an overview, e.g. [Gus97, Wat95].

Here, we consider problems that involve approximate pattern matching. Searching specific short sequences in long sequences (e.g. a genome at large) appears as basic operation in many problems related to the field of computational biology. Such operations can obviously be modeled as searching a pattern in a text. However, in most cases exact pattern matching is of little use in biological problems since genetic sequences, even among two members of the same species, show natural variances – nevertheless they are very similar. Most commonly, sequence comparison involves the application of some *alignment* algorithm. Alignment generally means lining up sequences allowing for mismatches, matches, and introduced gaps – which correspond to insertions or deletions. However, finding the optimal alignment of two sequences is a demanding task. Thus, in some applications where we can bound the number of errors, that is we can limit the degree of tolerated difference, approximate pattern matching can be used to increase the efficiency significantly, especially by using indices.

We briefly give some examples of biological problems where it is convenient to restrict the number of errors.

1.2.1 Examples from Computational Biology

- **STS and EST Maps**

  We illustrate this by considering the problem of finding sequence-tagged-sites (STS's) in newly determined genomes [Gus97]. Without going into biological detail, an STS is, from a simplified point of view, a DNA string consisting of 200-300 nucleotides whose left and right ends of length 20-30 occur only once in the entire genome. Thus, each STS occurs uniquely in a DNA of interest. One of the early goals of the Human Genome Project [Coo94] was to select and locate a set of STS's so that any substring in the genome of length $100,000$ contains at least one STS. Having such a map, one can use it to locate any sufficiently long string of anonymous, but sequenced DNA. Of course, there might appear errors in the map or the newly sequenced portion of DNA, either by natural variance or sequencing errors, but the number of errors should be a small percentage of the length of the STS, so fast approximate pattern matching methods can be used.

  A similar problem is the search for expressed sequence tags (EST's) in newly sequenced DNA. EST's are obtained from mRNA and represent protein coding parts of genes. Note that STS's might correspond to intergene parts of a DNA sequence.

- **DNA assembly**

  Chang and Lawler [CL94] point out that approximate pattern matching is important in the field of DNA assembly methods. The sequencing of huge DNA stretches usually involves splitting the DNA in many overlapping short sequences. The reassembly of the DNA solves a massive number of instances of the *approximate suffix-prefix matching problem*. Those computations have accounted for about 90 % of the total computing time in sequence assembly projects [Gus97] using standard alignment algorithms. However, in this case the only matches of interest are those with a small number of errors, so faster approximate pattern matching algorithms can speed up the assembly significantly.

- **Gene Polymorphism**

  The analysis of genes that possibly cause or contribute to certain genetic diseases gives also an example of an approximate pattern matching problem in computational biology. To identify such genes, copies of a suspected gene or region are sequenced from people who suffer from that disease and people (usually relatives) who do not. The sequences are compared in order to determine regions that might possibly contribute to the disease. Genetic diseases are usually caused by very small differences, so it makes sense to restrict the number of errors. More generally, consistent differences of genes (gene polymorphisms) are systematically investigated, and this is an active research area. A similar problem arises in the field of "molecular epidemiology" [Gus97].

- **Building and Maintaining Nonredundant Databases**

  In molecular biology it is often helpful to construct nonredundant databases. With such databases searches can be performed much faster and the statistical significance of matches can be evaluated better. Standard methods for maintaining such databases usually eliminate sequences that have high similarity to other sequences, so approximate pattern matching can be used to find such sequences. Considering the huge amount of sequence data available and the high rate at which this amount increases, efficient methods that help in creating and maintaining such databases are clearly welcome. We point out that especially when creating such a database, the number of sequence comparisons needed is quite large (all database entries against themselves) so that the use of efficient methods is quite important.

- **Approximate Codon Matching**

  The problem of approximate codon matching arises when comparing DNA and protein sequences. One is interested in finding subsequences of a DNA that translate to a specific amino acid sequence. Here, we have that the text t and the pattern p are from different alphabets (nucleotides and amino acids), but there is a mapping from nucleotide-triplets to amino acids. However, this mapping is not one-to-one since different *codons* may translate to the same amino acid. Additionally, there might appear framing errors caused by insertions or deletions. Chang and Lawler [CL94] have shown that this problem can be solved with an approximate pattern matching algorithm since the number of errors can be bounded when considering all possible reading frames and codons.

1.2.2 Examples from Other Fields

As mentioned above, other areas of application for this problem are signal processing and text retrieval.

The area of signal processing is very active. With an increasing number of applications using multimedia databases, it becomes more and more important to retrieve by content in audio, video, or image data. Perfect matches are almost impossible, so fast and space efficient approximate matching methods are of great importance.

In text retrieval the problem of finding or correcting misspelled words in texts is a classic application of approximate pattern matching. Damerau [Dam64] points out that 80 % of such errors are corrected by allowing just one substitution, insertion, deletion, or transposition.

There is a huge, constantly growing number of areas that are in demand of efficient approximate pattern matching methods. Just to mention a few, we name handwriting recognition [LT97], image compression [LS97], information retrieval [BYRN99], and file comparison [Hec78]. For other applications we refer to [Nav01, SK83].

1.3 Previous Work

Gabriele et al.

Gabriele, Mignosi, Restivo, and Sciortino [GMRS03, GMRS02] have recently proposed an index structure for approximate pattern matching. In their approach k errors are allowed in every window of size r, i.e. every substring of the pattern of length r, where r can be a constant or a function of the text length. The running time of their search algorithm is $O(m+occ)$ on average, where m is the length of the searched pattern and occ is the number of occurrences. The size of their index structure is $O(n \cdot polylog(n))$ on average, where n is the length of the considered text. The idea of this method is to consider "local" *languages* of strings that are in the *neighborhood* of any substring of the text. A string α is said to be in the neighborhood of string β with respect to k and r if it matches β with at most k errors in every window of size r and has the same length as β. Observe that any pattern that has an approximate occurrence in the text belongs to such a language. Moreover, instead of considering the whole languages, which might have a rather large size, only a subset of the languages is used to build the index structure.

An important concept is the *repetition index* of a text t with respect to k and r, denoted $R(t,k,r)$. $R(t,k,r)$ is the smallest value of h so that every substring of t of length h occurs at most in a unique position in t up to k errors for every r characters. That is, if the length of a pattern is greater than or equal to the repetition index, this pattern has at most one approximate occurrence in t. This observation leads to an easy solution for the case that the pattern is large.

For small patterns , i.e. $m < R(t,k,r)$, a set of data structures is used that enables to reduce the approximate pattern matching problem to an instance of the *document listing problem*, which is solved by applying a solution by Muthukrishnan [Mut02]. More specifically, the text is divided into overlapping fragments, and for each fragment a document is constructed consisting of strings that are in a certain "local" language referring to the fragment. The length of the fragments is in the first place dependent on the repetition index of t, and in an iterative process additional data structures for fragments of smaller lengths are constructed. Those data structures are stored in a stack, and for any input pattern the data structure with fragments of the appropriate length is chosen to answer the approximate pattern matching query.

In the average case analysis the underlying probabilistic model is that the text t is a prefix of an infinite sequence created by a memoryless source with symmetric symbol probabilities $p_i = 1/|\Sigma|$.

The results that lead to the average case complexities mentioned above can be summarized as follows.

- For the large pattern case the automaton used only needs to answer if the pattern has an approximate occurrence in the text or not. The size of this automaton is on average $O(n \cdot polylog(n))$. Note, however, that the "polylog" can be quite large. The largest exponent is mainly dependent on k and $R(t,k,r)$.

- The repetition index is on average $O(\log n)$, which leads to an efficient average case algorithm for finding the repetition indices. The total time needed by this algorithm dominates the preprocessing time.

- The sizes of the documents for the overlapping fragments can be bounded on average and converge to $O(n \cdot polylog(n))$ in the iterative process.

The analysis is only valid for the case $\frac{k}{r} < \frac{1-p_i}{2}$. Additionally, the running time of the search algorithm is dependent on the following assumption. Let R' be the expected value of $R(t,k,r)$ for a text of length n. Let further be $\beta > 1$. Then the analysis of the search time is only valid if the probability that $R(t,k,r)$ is greater than $R' \cdot \beta$ converges to zero faster than $\frac{1}{n}$. This assumption is conjectured to be realistic since large deviation theorems in similar settings state that the speed of convergence of variables representing the average of quantities is quite fast.

Amir et al.

Another proposal for an index structure was made by Amir et al.[AKL$^+$99]. Their approach only handles the case $d = 1$. That is, with this index structure it is only possible to locate substrings of the text that match p with one error. The size of the structure is in the worst case $O(n \cdot \log^2 n)$, and the look-up time is $O(m \cdot \log n \cdot \log\log n + occ)$. Buchsbaum et al. [BGW00] improved this result to $O(n \cdot \log n)$ and $O(m \cdot \log\log n + occ)$.

Their basic idea is to use suffix trees T for t and T^{-1} for the reverse text t^{-1}. In order to find a substring of t that matches p with one error, say at position j, they look for nodes in T and T^{-1} that are labeled $p(j+1,m)$ and $p(j-1,1)$ respectively. Observe that an occurrence of p in t at location i with error at position $i+j$ implies that $t(i, i+j-1)$ matches $p(1,j)$ and $t(i+j+1, i+m-1)$ matches $p(j+2,m)$. Observe also that all substrings of t and t^{-1} that match the mentioned portions of p are stored in the subtrees of these two nodes. Thus, if such nodes exist, it has to be verified that there is an intersection of leaves of the two subtrees so that there is a continuous substring $t(i, i+m-1)$ that matches p with one error at position j. To achieve this, the leaves in T are labeled with the starting location of the corresponding suffix, and the leaves in T^{-1} are labeled $n-i+3$, where i is the starting location of the corresponding suffix of t^{-1}. Hence, if two leaves of T and T^{-1} have the same label, two segments of t separated by one character are found, and an occurrence of p with one error is reported.

The bottleneck of this approach is to find the intersection of leaves of two subtrees for each possible error location j. In the original paper Amir et al. reduced this problem to the well-studied *range searching on a grid problem*. In this problem one is given n points in a 2-dimensional space and asked to report all points that lie in a specified rectangle. They use a solution by Overmars [Ove88] that preprocesses the points in time and space $O(n \log n)$ and answers queries in time $O(occ + \sqrt{\log n})$. The additional factors come from handling the case of exact matches. For details we refer to the original paper. The improvements in [BGW00] also address this problem.

Navarro and Baeza-Yates

Navarro and Baeza-Yates [NBY99] combine suffix trees and pattern partitioning. The pattern is partitioned and searched in the suffix tree with less errors. The occurrences are then verified for a complete match. The cost of the search in the suffix tree with errors grows with the size of subpatterns, but small sized subpatterns increase the cost of verifying matches. For searching the suffix tree with errors, Navarro and Baeza-Yates use a *bit-parallel automaton*. The basic idea for traversing the suffix tree is to use a limited depth first search starting at the root and stopping when it can be seen that the current path cannot start an approximate substring. More specifically, this can be done by computing the edit distance between the current path and the pattern and reporting the leaves as results if at some point the edit distance is $\leq k$ (where again

k is the number of allowed errors). On the other hand, it can be determined if a path cannot possibly be continued to have an edit distance as low as k for stopping the search. Observe that at depth $m + k + 1$ the search can surely be abandoned. The approach discussed here simulates this traversal by using a bit-parallel automaton. We refer to the original articles [NBY99] and [NBY00] for details. Their method for optimally splitting the pattern is based on the partitioning techniques of [Mye94, BYN99]. The search time is $O(n^\lambda)$, $\lambda < 1$ for a moderate number of errors. In particular, the error level $\alpha = k/m$ must satisfy $\alpha < 1 - e/\sqrt{|\Sigma|}$, where $|\Sigma|$ is the alphabet size.

Comparison to our Results

Comparing our newly developed index structure to the existing solutions, we note that there is no other method with reasonable space requirements that has a linear search time with respect to the pattern size in the worst case. The look-up time grows with n in the methods of both Amir et al. and Navarro and Baeza-Yates. Gabriele et al. achieve our search efficiency only in the average case. However, up to now our approach is limited to the one error case and to the *Hamming distance*, but we hope to be able to extend the method. Another problem is the size of our index structure, which might become quadratic in the length of the text. However, the probability of this is very low in commonly used probabilistic models. The average size of our structure is competitive to the other index structures having the same asymptotics as the structure of Amir et al. and even better asymptotics than Gabriele et al.. Additionally, our approach is much simpler than the previous proposals and seems, in contrast to the method of Amir et al., to be extendable to handle more errors.

Chapter 2

Basic Data Structures and Algorithms

In this chapter we give basic definitions and data structures which are essential to considering pattern matching problems. Additionally, we describe algorithms that are needed by our method for approximate pattern matching. Throughout this work when analyzing data structures and algorithms, we restrict ourselves to the *uniform cost model*.

2.1 Basic Definitions

Definition 2.1 (Strings). *Let Σ be an arbitrary, finite alphabet. A string $t = x_1 x_2 \ldots x_m$ is an ordered list of characters $x_i \in \Sigma$. Σ^* is defined as the set of all finite strings over characters of the alphabet Σ, which also includes the empty string, denoted ϵ. $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$ is the set of all nonempty strings over Σ. The length of a string $t = x_1 x_2 \ldots x_m$ is defined as its number of characters and is denoted by $|t| = m$. For any string t, $t(i,j), 1 \leq i \leq j \leq m$, is the (contiguous) substring of t starting at position i and ending at position j, $x_i x_{i+1} \ldots x_j$. The substring $t(i,i)$, which is simply the character x_i, is denoted by $t(i)$. $t(i,j) = \epsilon$ for $i > j$.*

In this work we use the following notational concepts. We refer to strings with t and p (usually when talking about text and pattern) or with Greek lower case letters α, β, γ. Single characters are denoted by a, b, c, d, e, x. For strings $\alpha, \beta \in \Sigma^*$ we define $\alpha\beta$ as the concatenation of the two strings.

Definition 2.2 (Suffixes, Prefixes). *Let $t = x_1 x_2 \ldots x_m$ be a string of length m. $t(i,j)$ is a suffix of t if $j = m$. Conversely, $t(i,j)$ is a prefix of t if $i = 1$. We usually denote by suf_j the suffix beginning at position j, which equals $t(j,m)$. We call a suffix α of t proper if $\alpha \neq t$. Conversely, a prefix α of t is called proper if $\alpha \neq t$.*

For $\alpha, \beta, \gamma \in \Sigma^*$ and $t = \alpha\beta\gamma$, α is a prefix, β is a substring and γ is a suffix of t. A suffix α of t is called *nested* if there exists another suffix β of t so that α is a proper prefix of β. This means essentially that we have at least two different occurrences of α in t.

We now define *trees* over strings.

Definition 2.3 (Σ^+-Tree). *A Σ^+-Tree T is a rooted, directed tree with nonempty strings over the alphabet Σ as edge labels. For each character $a \in \Sigma$ a node v in T has at most one outgoing edge whose label starts with a.*

Let T be a Σ^+-tree and let v be a node in T. By $path(v)$ we denote the concatenation of all edge labels along the path from the root of T to the node v. By definition labels of outgoing edges of any node start with different characters, and hence path labels are unique. Analogously, for two nodes u and v where there is a path from u to v in T, $path(u, v)$ denotes the concatenation of edge labels on that path. For a string α, $path(\alpha)$ or the α-path is the path in T starting at the root node so that the concatenation of edge labels (and possibly a prefix of the last edge label on the path) spells out α.

By T_v we denote the subtree of T that is rooted at node v. Similarly, by T_α we denote the subtree of T that is rooted at the end of $path(\alpha)$. If there is no node v, $path(v) = \alpha$, we think of the root of the subtree as a newly inserted node after the last character of $path(\alpha)$.

T is *atomic* if every edge in t is labeled with a single character. T is *compact* if every node in T is either the root, a leaf, or a *branching node*. A branching node v is a node that has at least two outgoing edges.

Definition 2.4 (Words in Σ^+-Trees). *A string α occurs in T if and only if there is node v in T with $path(v) = \alpha\beta$ for some $\beta \in \Sigma^*$. By words(T) we denote the set of all strings occurring in T.*

Definition 2.5 (Σ^+-tree Built by Strings). *Let $\{\alpha_1, \ldots, \alpha_n\}$ be a set of strings. The Σ^+-tree that is built by these strings is the uniquely determined tree T with the property that $\alpha \in words(T)$ implies that α is a prefix of some α_i, $1 \leq i \leq n$.*

Definition 2.6 (Suffix Trees). *Let $t \in \Sigma^+$ be a string of length n. A suffix tree T for t is a Σ^+-tree with additional properties:*

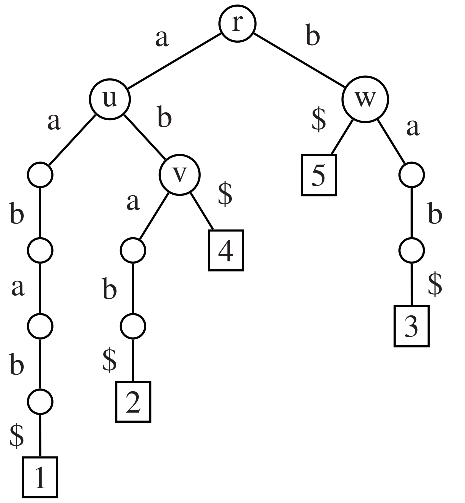
1. *T has exactly n leaves labeled from 1 to n*

2. *For any leaf i the concatenation of edge labels on the path from the root to leaf i equals suf_i of t. That is, that concatenation spells out $x_i x_{i+1} \ldots x_n$.*

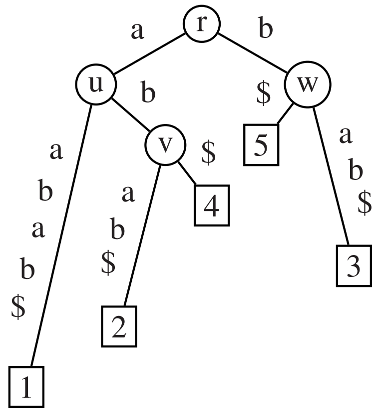
When talking about suffix trees in the following, we always refer to *compact suffix trees*, which we denote $CST(t)$. Similarly, when referring to Σ^+-trees, we refer to compact Σ^+-trees. An *atomic suffix tree* is often called *suffix trie* and we adopt this notation here. See Figure 2.1 for examples. If there exist substrings αa and αb of t, $a \neq b$, α is called a *right branching word*. Observe that if α is right branching word, there exists an internal node v in $CST(t)$ with $path(v) = \alpha$. This can be seen easily and is common knowledge (see, e.g. [Maa00]).

The above definition of suffix trees does not guarantee that a suffix tree really exists for a given string t. This problem arises if some suffix of t is a proper prefix of another suffix. In that case the first suffix does not end at an external node, so that the second condition is violated.

To avoid this, we add an additional character that is not part of Σ to the end of t. This character is usually denoted by $\$$ and called *sentinel* or *termination* character. By doing this, no suffix can be a prefix of another suffix. Throughout this work, unless stated otherwise, we assume that suffix trees for any text t are actually suffix trees for $t\$$ without explicitly stating that we add the sentinel character.



(a) atomic suffix tree or suffix trie for $aabab\$$



(b) compact suffix tree for $aabab\$$

Figure 2.1: A suffix trie and a compact suffix tree for the text $aabab\$$. The compact suffix tree results from the trie by eliminating all unary nodes

Definition 2.7 (Node Depth). *Let v be a node in a Σ^+-tree T. The node depth of v is defined as the number of nodes on the path from the* root *of T to v excluding the* root *and including v.*

Definition 2.8 (String Depth). *Let v be a node in a Σ^+-tree T. The string depth of v is defined to be $|path(v)|$.*

In some applications it is convenient to store the node depth and the string depth of a node v with this node. We briefly show how these values can be fixed in a linear preprocessing step. To achieve this, we traverse the nodes of T by a *depth first search* (*DFS*) starting at the root node. Obviously, the node depth of any visited node is equal to the recursion depth of the DFS traversal. This value can easily be memorized with the help of an index that is increased by 1 when moving forward and is conversely decreased by 1 when moving backward. To see how the string depths can be fixed, suppose that node v is visited and we know the string depth k of node u that is the parent of v. Let (i, j) be the label of (u, v). We refer to Section 2.3 for the explanation of how to store edge labels with a pair of indices. Then the string depth of v is clearly given by $k + (j - i + 1)$. At the beginning we set the string depth of the root to 0 and inductive arguments proof the correctness of this methodology. Thus, we have

Lemma 2.9 (Linear Determination of Node and String Depths). *Let T be a suffix tree. Storing the string and node depths of all nodes can be performed in time $O(n)$.*

Additionally, we introduce the *height* of T.

Definition 2.10 (Height of a Suffix Tree). *Let T be a suffix tree. The height h of T is the maximal node depth of any node $v \in T$.*

2.2 Problem Definition

We give a formal definition of the approximate pattern matching problem, see also [Nav01].

Definition 2.11 (Approximate Pattern Matching Problem). *Let Σ be a finite alphabet of size $|\Sigma| = V$. Let $t \in \Sigma^*$ be a text of length $n = |t|$. Let $p \in \Sigma^*$ be a pattern of length $m = |p|$. Let $k \in \mathbb{R}$ be the maximum error allowed. Let $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ be a distance function.*
The problem of approximate pattern matching is, given t, p, k, and d, to return the set of all text positions j so that there exists an i with $d(p, t(i, j)) \leq k$.

The distance $d(\alpha, \beta)$ between two strings can be considered to be the minimal cost of a sequence of operations to transform α into β and ∞ if no such sequence exists. The total cost is the sum of the individual operations. Before taking a closer look on the most common distance functions, we consider possible operations that can transform one string into another string. The operations are a finite set of rules of the form $\sigma(\alpha, \beta) = s$, where α and β are different strings and s is any nonnegative real number, i.e. the cost of transforming α into β by the considered operation. In most applications the set of such operations is restricted to the following local operations.

- **Insertion**: $\sigma(\epsilon, a)$, i.e. inserting the letter a
- **Deletion**: $\sigma(a, \epsilon)$, i.e. deleting the letter a
- **Substitution**: $\sigma(a, b)$, i.e. replacing or substituting letter a by letter b for $a \neq b$
- **Transposition**: $\sigma(ab, ba)$ for $a \neq b$, i.e. swap the adjacent letters a and b

Note that such a set of operations reflects the most common corruptions that for example texts or genomes are exposed to with the exception of block operations, which we do not consider here. Generally, the set of operations allowed along with their costs must be adopted to the underlying problem.

2.2.1 Distance Functions

We now introduce some of the most commonly used distance functions.

- **The Levenshtein or edit distance** [Lev65] allows insertions, deletions, and substitutions. In the simplified version the cost of each operation is 1. Thus, the distance between two strings can be considered as the minimal number of such operations that are necessary to transform one string into the other. The approximate pattern matching problem with this distance function as error model is often called *string matching with k differences*.

- **The Hamming distance** [Ham50] allows only substitutions. If one substitution can be performed at the cost of 1, the concerning approximate pattern matching problem is called *string matching with k mismatches*.

- **The longest common subsequence distance** [NW70] allows only insertions and deletions, all costing 1. Thus, this distance measures the length of the longest pairing of characters that can be made between both strings without altering the order of the characters. The distance is the number of unpaired characters. The name of this distance reflects this fact.

Observe that if the Hamming distance or edit distance is used with cost 1 per operation, the maximal tolerated error is restricted to be $< m$. Otherwise, any arbitrary substring of t could be transformed into p without violating the upper bound for the error.

In this paper we use the simple Hamming distance as error model. To reflect this, we say we look for substrings that match a given pattern with one mismatch. This distance is a simplification of the more common Levenshtein distance. However, even in this simplified version, building an index for approximate pattern matching is still a demanding task, and we believe that efficient solutions for the easy case can expedite solutions for the more complicated cases.

2.3 Properties of Suffix Trees

It follows from the definitions of compact Σ^+-trees and suffix trees that every suffix tree for a text of length n has $O(n)$ nodes since every internal node is branching and there are n leaves. The number of edges is hence also bounded by $O(n)$ as we deal with trees. The edge labels, however, consist of substrings of t, and the size required for storing them explicitly is linear in the length of that substring. In order to avoid this, we store only indices i and j at an edge if its edge label is given by $t(i, j)$. Thus, the size needed to store any edge label is constant and the following lemma holds.

Lemma 2.12 (Size of Suffix Trees). *The size of a compact suffix tree for a text t with length n is $O(n)$.*

Because of the importance of *suffix links* both in construction of suffix trees and many applications, we define them formally.

Definition 2.13 (Suffix Links). *Let $x\alpha$ be an arbitrary string. Let v be an internal node in a suffix tree T with $path(v) = a\alpha$. Let w be a node such that $path(w) = \alpha$. Then a pointer from v to w is called the suffix link of v, denoted suffixlink(v).*

The following classic lemma gives suffix trees a nice structure. The proof is obvious.

Lemma 2.14 (Weiner76). *Let T be $CST(t)$ and let $a \in \Sigma$, $\alpha \in \Sigma^+$. If there exists a node u in T so that $path(u) = a\alpha$, then there also exists a node v in T with $path(v) = \alpha$.*

Note that according to Lemma 2.14, suffix links exist for any internal node v. As a special case, if $\alpha = \epsilon$, the suffix link of v points to the root.

2.4 Construction of Suffix Trees

There are several algorithms that construct compact suffix trees in linear time. Weiner [Wei73] presented the first such algorithm and showed that this can be done in linear time. McCreight [McC76] and Ukkonen [Ukk95] introduced methods that are far less space demanding. We briefly present Ukkonen's algorithm for the construction of suffix trees, which is conceptually the easiest of them. Furthermore, his algorithm works *on-line*, that is we read text t character by character and obtain an *implicit suffix tree* for $t(1,i)$ after having processed character $t(i)$. This property has shown to be useful in certain situations.

Before considering Ukkonen's algorithm, we define implicit suffix trees.

Definition 2.15 (Implicit Suffix Trees). *An implicit suffix tree for the text* t *is a* Σ^+*-tree so that* $words(T) = \{w|w$ *is a substring of* $t\}$.

An implicit suffix tree for t is obtained from the suffix tree for $t\$$ by removing every copy of the sentinel character $\$$ from the edge labels, then removing any edge that has no label, and then removing any node that is not branching. In other words, an implicit suffix tree is like a suffix tree as given by Definition 2.6 but without the restriction of having n leaves. Obviously, the on-line property of a construction algorithm can only be achieved with implicit suffix trees. Although an implicit suffix tree encodes all suffixes of t, it is less informative since there is no marker that indicates the end of such a path. However, the implicit suffix tree for t resulting from Ukkonen's algorithm can easily be converted to a suffix tree for $t\$$. Throughout this description of Ukkonen's algorithm, we refer to implicit suffix trees when talking about suffix trees.

For the sake of simplicity we start by considering the construction of a suffix trie. As mentioned above, we intend to construct it on-line. Let T_i be the trie for $t(1,i)$. Then we construct T_{i+1} from T_i by extending each suffix in T_i by character $t(i+1)$ if necessary. Thus, we have to locate the nodes that correspond to the suffixes $suf_1, \ldots, suf_i$ and append a new edge labeled with character $t(i+1)$. The naive method to locate the appropriate nodes is to search them starting from the root, which takes time $O(|suf_j|)$. Hence the total construction time of T_{i+1} from T_i is $\sum_{j=1}^{i} O(|suf_j|) = O(i^2)$, and this leads to a construction time of $O(n^3)$ for T_n.

We recall the definition of suffix links. Let v be the node that represents suf_1. From there we reach the node w that represents suf_2 by following the suffix link of v. Thus, by memorizing the longest suffix after each iteration, we can easily visit all other suffixes and lengthen them by the newly read character. However, we have to assure that we can build the suffix links along with the construction of the suffix trie. We show that this is possible when discussing the subject in the context of suffix trees. Analysis of this approach reveals a running time of $O(n^2)$. It can be shown to be optimal.

We now return to the construction of suffix trees. The basics of the algorithm remain the same. However, we have to introduce some new concepts. We give a high-level description of Ukkonen's algorithm.

Algorithm 1 Ukkonen High-Level

Construct T_1
for i from 2 to n **do**
  for j from 1 to n **do**
    Find the end of the path from the root to $t(j, i-1)$, that is, suf_j of iteration $i-1$. If necessary, add character $t(i)$ to extend the path so that $t(j, i)$ is in T_i
  end for
end for

If we consider the high-level description of the algorithm, we see that it is conceptually very easy and does not differ from the method for constructing tries. In the following we call one passing of the outer for-loop an iteration, whereas the inner for-loop refers to an extension, e.g. extension j of iteration i. Moreover, when talking about suf_j in iteration i, we mean $t(j, i)$, which corresponds to suf_j in T_i. Again, the naive implementation turns out in a $O(n^3)$ construction time. However, we show that we can find those suffixes in amortized constant time as well as that we need only amortized constant time for each iteration by applying a few observations and implementation tricks.

2.4.1 Traversing the Suffixes

As mentioned above, we cannot afford to locate the end of $t(j, i-1)$ in extension j of iteration i by searching it in the current tree. Instead, we want this work to be done in amortized constant time.

Before describing how we can achieve this by using suffix links, we introduce the concept of *reference pairs*.

Reference Pairs

In compact suffix trees there are no unary nodes. To be able to work with such implicit positions nevertheless, we use reference pairs. A reference pair consists of a node and some word β and is denoted by (v, β) so that $\alpha = path(v)\beta$ if (v, β) is a reference pair for α. As β corresponds to some substring of t, we refer to it with a pair of indices (pos, len) if $\beta = t(pos, pos + len - 1)$. Since we use reference pairs to access the position of a substring in the current tree, we want to choose node v of the reference pair $(v, (pos, len))$ such that it is as close to that position as possible. We call a reference pair $(v, (pos, len))$ for a given substring the *canonical reference pair* if there is no other reference pair $(v', (pos', len'))$ with $len' < len$.

Using Suffix Links

Suppose now that we have just extended suf_{j-1} in iteration i and want to proceed with extending suf_j. Let $suf_{j-1} = a\alpha\beta$. Suppose $(v, (pos, len))$ is the canonical reference pair for suf_{j-1} and $path(v) = a\alpha$. Let now w be the node to which the suffix link of v points. Clearly, by Lemma 2.14 such a node exists. We have $suf_j = \alpha\beta$ and $path(w) = \alpha$. Thus, we have to follow a path labeled β beginning at node w in order to access suf_j. Let $|\beta| = len$. Then, $(w, (pos, len))$ is obviously a reference pair for $\alpha\beta$. After making $(w, (pos, len))$ a canonical reference pair, we can access suf_j in time $O(1)$.

It remains to show that for any internal node v there exists the suffix link of v in T_i. We address this question below when specifying the complete algorithm. The above is a description of how we traverse the suffixes for which we need explicit work in each iteration. Procedure *Canonize*, which makes a reference pair canonical, just jumps from node to node along the edges whose labels start with the appropriate character. This can be done as we know that the path along which we move surely exists in the tree.

Algorithm 2 Canonize

Input: A reference pair (u,(pos,len))
Let (u, v) be the edge whose label starts with $t(pos)$.
$l \leftarrow length((u, v))$
while $l \leq len$ **do**
  $u \leftarrow v$
  $len \leftarrow len - l$
  $pos \leftarrow pos + l$
end while

2.4.2 Suffix Life Cycle

We now characterize suffixes in a certain iteration i. The following is derived from observations about the way that a suffix suf_j can be affected by successive extensions and iterations. We divide all suffixes in iteration i into 3 classes.

- **Class 1:**
  Suffixes that are not a prefix of another suffix (non-nested suffixes).

- **Class 2:**
  Suffixes that are a prefix of another suffix in iteration $i - 1$, that is $t(j, i - 1)$ is a prefix of some $t(k, i - 1)$, but that are not a prefix in iteration i. We call such suffixes *relevant suffixes*.

- **Class 3:**
  Suffixes that are a prefix of another suffix in both iteration $i - 1$ and iteration i.

We now show that only suffixes in class 2 induce work by applying a simple implementation trick. Consider a suffix suf_j in class 3. Clearly, there has nothing to be done as there already exists a path $t(j, i)$ in T_i.

Consider now a suffix suf_j in class 1. Since $t(j, i)$ is non-nested in iteration i, extending suf_j cannot cause this suffix to be a prefix of another suffix in some iteration $i + l$. Suppose for the sake of contradiction that there exists a suffix suf_k in iteration $i+l$ for which suf_j is a prefix, that is $t(j, i+l)$ is a prefix of $t(k, i+l)$. Then obviously $t(j,i)$ is a prefix of $t(k,i)$, which contradicts the fact that suf_j is non-nested in iteration i. Thus, if a suffix suf_j is in class 1, it remains to be in class 1 for the rest of the algorithm. Nevertheless, those suffixes have to be extended in each iteration. To avoid explicit work for this, we introduce so-called *open edges*. An open edge goes into a leaf node, and the label of the edge is a suffix of the currently considered word $t(1,i)$. So instead of increasing the right index of all open edges in each iteration, we just memorize the left index and that the edge is open. Open edges are labeled (j,∞) and are extended automatically. Thus, the total time needed to extend all suffixes in class 1 in iteration i is constant.

Consider now the following suffix cycle. In every iteration i $suf_i = t(i)$ is added to T_i. If we consider suf_i to be the empty word in iteration $i-1$, suf_i was surely nested in that iteration and was therefore in class 3. At some point suf_i will be extended so that it is no longer nested. Thus, suf_i is then in class 2 and from the next iteration on it will remain in class 1.

Suffixes in class 2 are the only suffixes that induce explicit work. A new internal node has to be created along with its suffix link and a new open edge. Therefore, we call them relevant suffixes. As a suffix is in class 2 in at most one iteration of the algorithm, this indicates how we prove that the number of suffixes that induce explicit work in each iteration is amortized constant.

2.4.3 Putting it Together

We start by giving a simple fact about nested suffixes.

Lemma 2.16 (Active Suffix). *Let suf_j be a suffix that is nested in iteration i of Ukkonen's algorithm. Then all shorter suffixes suf_k, $j \leq k \leq i$ are also nested in iteration i.*

Proof. Consider an arbitrary suffix suf_k where $j \leq k \leq i$. Let suf_l be a suffix so that suf_j is a prefix of suf_l. Then $t(l, l+m-1) = suf_j = t(j,i)$, where $|suf_j| = m$. But we also have $suf_k = t(k,i) = suf_j(k,i) = t(l+(k-j), l+m-1)$, and thus suf_k is a prefix of $suf_{l+(k-j)}$. □

We call suf_j the *active suffix* if it is the longest nested suffix after iteration $i-1$. The active suffix is the starting point of looking for relevant suffixes in iteration i. Recall that a suffix that is not nested in iteration $i-1$ can never be nested in iteration i.

After each iteration we memorize the longest nested suffix by a reference pair. In the next iteration we check if the active suffix is still nested. If so, we update the reference pair. If not (class 2), we introduce a node at the old position and an open edge for the suffix that will now stay non-nested for the rest of the algorithm. Then we replace the node v of the reference pair by suffixlink(v), so that we obtain a reference pair for suf_{j-1}. We make the new reference pair canonical and continue the procedure for suf_{j-1} until we have found a suffix that remains nested (class 3). By Lemma 2.16 we can then turn to the construction of T_{i+1}. This suffix will be the active suffix at the beginning of the next iteration.

We can now specify the complete algorithm for one iteration of Ukkonen's algorithm.

Algorithm 3 An Iteration of Ukkonen's algorithm

```
Input: T_i, t, (u, (pos, len)) so that this is the canonical reference pair for the active suffix
v ← testandsplit((u, (pos, len)), t(i + 1))
p ← ⊥
while v ≠ ⊥ do
   w ← new leaf
   Join v and w by a new open edge (i + 1, ∞)
   if p ≠ ⊥ then
      suffixlink(p) ← v
   end if
   p ← v
   u ← suffixlink(u)
   Canonize(u, (pos, len))
   v ← testandsplit((u, (pos, len)), t(i + 1))
end while
if p ≠ ⊥ then
   suffixlink(p) ← u
end if
len ← len + 1
Canonize(u, (pos, len))
```

We now show that the suffix links of internal nodes are constructed by the algorithm.

Lemma 2.17 (Existence of Suffix Link Nodes). *If a new internal node v is created in extension j of iteration i, either the node suffixlink(v) already exists or it will be created in extension $j+1$ of the same iteration i.*

Proof. Let $path(v) = a\alpha = suf_j$. A new internal node is only created if suf_j is in class 2 in iteration i. The path $a\alpha$ in T_{i-1} then continues with some character different from $t(i)$, say b. Consider now $suf_{j+1} = \alpha$ in the same iteration. Path α has certainly a continuation with character b. So either it only continues with b, or it also continues with some other character. In the first case, a new internal node w will be created in extension $j+1$ since $t(i) \neq b$. Otherwise, we already have an internal node w with $path(w) = \alpha$. □

Corollary 2.18 (Construction of Suffix Links). *In Ukkonen's algorithm, each internal node will have a suffix link by the end of the next extension.*

Proof. We give an inductive proof which obviously holds for T_1. Suppose the claim is true through the end of iteration i. Consider now the last extension of iteration $i+1$. In the last extension we find the longest suffix that is still nested, that is there is no new internal node created. Thus, by Lemma 2.17, all suffix links for newly created nodes are known and set correctly by Ukkonen's algorithm. □

Thus, for every created internal node we have its suffix link at hand by the end of the next extension. If a new internal node is created, we have at least one more extension in the same iteration. The suffix link that is used to access the suffix of this next extension is obviously not the suffix link of the newly created node. Hence, the suffix links are available when needed.

Before formally proving that the amortized time complexity of Ukkonen's algorithm is $O(n)$, we give the intuition of the proof. Firstly, the number of suffixes in class 2 in each iteration is amortized constant. This follows from that fact that each suffix suf_j can be in class 2 only in one iteration. Thus, it remains to consider the time needed for traversing the suffixes that induce explicit work. The time for that is obviously dominated by the total time needed by procedure *Canonize*. Consider a reference pair (v, β) for a relevant suffix. If this pair has to be made canonical, then obviously the length of β is decreased by at least one in each while-loop of the procedure. Each while-loop corresponds to moving one node down the path. In iteration 1, $|\beta|$ is obviously 1. Then in each iteration $|\beta|$ is increased by only 1 when the reference pair for the longest nested suffix of iteration $i-1$ is made the reference pair for the active suffix in iteration i.

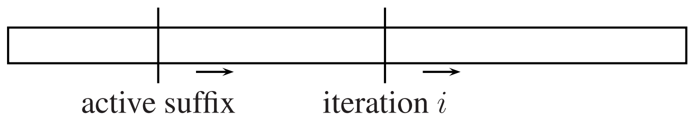


Figure 2.2: Illustration of the amortized linear time complexity of Ukkonen's algorithm. Both i and the index of the active suffix move to the right. In each iteration the number of relevant suffixes is proportional to the number of positions that the active suffix moves.

Theorem 2.19 (Complexity of Ukkonen's Algorithm). *The construction of a suffix tree for text t of length n by Ukkonen's algorithm takes time $O(n)$.*

Proof. We have obviously n iterations since in every iteration exactly one character is processed. Procedure *testandsplit* needs only a constant amount of operations if the reference pair is canonical. Furthermore, the while-loop is in total traversed only n times since in each traversal a new leaf node is added and we have n leaves.
Consider now the call of procedure *Canonize*. Outside the while-loop *Canonize* is called exactly once in each of the n iterations. Additionally, it is called once in a traversal of the while-loop. Thus, we have at most $2n$ calls of *Canonize*. The time complexity of *Canonize* is determined by the number of times its while-loop is traversed. Observe that in each traversal of its while-loop len is decreased by at least 1. In contrast to that, len is increased by 1 exactly once in each iteration. Since we have n iterations, we have at most n traversals of the while-loop of procedure *Canonize*. Thus, we have totally $O(n)$ operations. □

Corollary 2.20 (Linear Suffix Tree Construction). *A suffix tree for text t of length n can be constructed in time $O(n)$*

2.5 Exact Pattern Matching

Exact Pattern Matching is the classic application for suffix trees. Let $T = CST(t)$. We consider the situation where a text t of length n is fixed and several patterns p of length m are compared with the text. The problem is the following. Determine if there are substrings $t(i, i+m-1)$ of t with $t(i, i+m-1) = p$ or not. If there are such occurrences, return the locations, that is the starting positions of these substrings. We show how to solve this problem in time $O(m+occ)$ for any pattern p, where occ is the number of substrings of t that match p exactly, by using a suffix tree for t.

The concept of that search is simple. We start at the root of T and match the pattern p character by character against a path in T. If we arrive at a node u so that $path(u) = p(1, i)$, then we continue our search with the edge (u, v), where the label of (u, v) starts with character $p(i+1)$ if such an edge exists. If at some point $p(i+1)$ cannot be matched, either against the next character of an edge label or against the first character of any outgoing edge label, there is no occurrence of p in t. On the other hand, when we have matched the whole of p, the starting positions of all substrings that match p exactly are given by the leaf labels in T_p. It is easy to see that we can collect the leaves in this subtree in time $O(occ)$ by DFS, where occ is the number of leaves in T_p. The time needed for finding p in T is obviously bounded by $O(m)$. We compare each character $p(i)$, $1 \leq i \leq m$, exactly one time with a character on some edge label in T. If we match inside any edge, there is no additional work. After arriving at a node, however, we have to find the appropriate outgoing edge. The number of outgoing edges is bounded by $|\Sigma|$, and for each such edge we just have to compare the first character of its edge label with the current character $p(i)$. This leads to the following theorem.

Theorem 2.21 (Exact Pattern Matching with Suffix Trees). *By using a suffix tree for text t, all occ substrings $t(i, i+m-1)$ of t that match a given pattern p of length m exactly can be found in time $O(m + occ)$.*

2.6 Lowest Common Ancestors

We consider the problem of finding the *lowest common ancestor* (lca) of two nodes in a tree. This fundamental algorithmic problem arises as a subproblem in many other problems and is inherently algorithmically beautiful. In this work we need this result in Section 3.5 in the context of developing an efficient solution for the Σ^+*-tree merging problem.* The surprising result that lca-queries can be solved in constant time with only linear preprocessing was first obtained by Harel and Tarjan [HT84]. It was later simplified by Schieber and Vishkin [SV88] and again by Bender and Colton [BFC00]. Here, we present the latter algorithm as it is by far the simplest optimal algorithm on this problem and easy to understand.

We begin by giving basic definitions according to [Gus97].

Definition 2.22 (Ancestor). *In a rooted tree T, a node u is an ancestor of node v if u is on the unique path from the root to v. Note that by this definition a node is an ancestor of itself.*

Definition 2.23 (Lowest Common Ancestor). *In a rooted tree T, the lowest common ancestor of two nodes u and v is the deepest node in T that is an ancestor of both u and v.*

The lowest common ancestor problem is to preprocess T such that lca-queries can be answered as quickly as possible. Both the preprocessing and the query time have to be considered and optimized.

Reducing lca to rmq

The basic idea of this simple solution is to reduce the lowest common ancestor problem to another problem called the *range minimum query problem*.

Definition 2.24 (Range Minimum Query Problem). *Let A be a length n array of numbers. A range minimum query for two indices i and j, $rmq(i, j)$, returns the index of the smallest element in the subarray $A[i, \dots, j]$.*

Before showing how to reduce the lca-problem to the rmq-problem, we consider the following. Suppose that we visit the nodes of a tree T by *DFS* and that we want to determine the lowest common ancestor of two nodes u and v. Obviously, the lowest common ancestor of both nodes is the node with the lowest node depth that is encountered (by forward or backward movements) between u and v are visited. To see this, recall the recursive nature of *DFS*. Thus, we have the following lemma.

Lemma 2.25 (LCA by DFS). *The lowest common ancestor of two nodes u and v in a tree T is the node with the smallest node depth that is encountered between the visits of nodes u and v by a depth first search.*

We define the *Euler Tour* of a tree T.

Definition 2.26 (Euler Tour of a Tree). *The Euler Tour of a tree T is the sequence of nodes in the order in which they are visited by DFS. Here, we consider each node to be visited by both forward and backward edges.*

Observe that this sequence has length $2 \cdot n - 1$, where n is the number of nodes in T. This follows from the fact that a tree has $n - 1$ edges and each of the edges is traversed exactly twice, once in each direction.

Obviously, an array $E[1, 2n - 1]$, where $E[i]$ is the i-th node on the Euler tour of T, can be determined in time $O(n)$ with a simple *DFS*.

We create two additional arrays $L[1, \dots, 2n - 1]$, where $L[i]$ holds the node depth of node $E[i]$, and $R[1, \dots n]$, where for each node u, $R[u]$ is the index of the first occurrence of node u in E. Both arrays can obviously be computed in linear time.

Now we show how we can answer a lca-query using these arrays and performing a rmq-query. We note the following

- All nodes that are visited between the discovery of the two nodes u and v during a *DFS* are $E[R[u], \dots, R[v]]$. Here, we suppose without loss of generality that u is discovered before v.

- The node depths of these nodes are $L[R[u], \ldots, R[v]]$. Thus, the node with the lowest node depth that is encountered in this sub-tour is $rmq(R[u], R[v])$, where the rmq-query is performed on L.
- According to Lemma 2.25 and to the properties of the used arrays, the lowest common ancestor of u and v is $E[k]$, where k is the result of the mentioned rmq-query. Recall that the rmq-query returns an index.

This leads to the following lemma which is proven by the above discussion.

Lemma 2.27 (LCA to RMQ Reduction). *Let u and v be two nodes with labels i and j, where $1 \leq i, j \leq n$. Assume that $R[i] < R[j]$. Let k be an integer such that $R[i] \leq k \leq R[j]$ and $L[k]$ is the minimum in $L[R[i], \ldots, R[j]]$. Then $E[k]$ is the lowest common ancestor of u and v.*

Thus, we have shown how to answer lca-queries with rmq-queries. Obviously, the above arrays can be determined in time and space $O(n)$. We continue with showing that we can answer rmq-queries in constant time with linear preprocessing. Hence, preprocessing the array L that is of size $O(n)$ for rmq-queries leads to the desired result.

Solving RMQ Queries

We begin with illustrating a solution to the rmq-problem for arrays of arbitrary numbers that has preprocessing complexity $O(n \log n)$. Then we show that the array L that we need to preprocess for the lca-queries has an additional property we can use to get a linear solution.

The naive way to preprocess an array for constant rmq-answers is to compute the rmq for each pair (i, j). The complexity of this approach is $O(n^2)$ using a dynamic programming approach to compute the answers.

Instead of using this brute force method, we precompute for each index i only the answers for ranges that have a length which is a power of 2. That is, for all i, $1 \leq i \leq n$, and for all $j, 1 \leq j \leq \log(n-i)+1$, we compute $rmq(i, i+2^j-1)$ and store these values in a table M. Thus, $M(i, j)$ contains the minimum element of the sub-array $A[i, i+2^j-1]$. The table size is $O(n \log n)$. In order to achieve a construction time that is linear in the size of this table, we use the following dynamic programming approach.

$$M[i,j] = \begin{cases} M[i, j-1], & A[M[i,j-1]] \leq A[M[i+2^{j-1}-1, j-1]] \\ M[i+2^{j-1}-1, j-1] & \text{otherwise} \end{cases}$$

Indeed, the minimum of the range $[i, i+2^j-1]$ is the smaller one of the minima of the two ranges. Thus, the table can be filled in linear time. Suppose now that we have precomputed these values and want to answer $rmq(i, j)$. To achieve this, we select two overlapping blocks that cover the whole of the considered range. Let $k = \lfloor \log(j-i) \rfloor$. Then we compare the minima of $M(i, k)$ and $M(j-2^k+1, k)$ to obtain the minimum of the whole range. Obviously, these two blocks are overlapping, and thus the lower of the two minima is the result of the range minimum query. Since we have precomputed the two minima of the blocks, just one comparison is needed and the rmq is answered in constant time.

We call this approach the *Sparse Table Algorithm*.

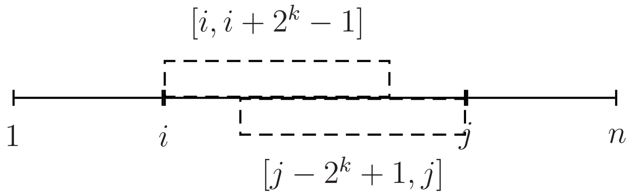


Figure 2.3: Choosing overlapping blocks of size 2^k for answering $rmq(i, j)$

Lemma 2.28 (Complexity of the Sparse Table Algorithm). *The preprocessing time of the Sparse Table Algorithm for an array of size n is $O(n \cdot \log n)$, and the query time is $O(1)$.*

Linear Preprocessing Solution

Consider again the array L. Recall that this array stores the level values of the nodes in the order implied by the Euler Tour. Obviously, the levels of two nodes u and v that are adjacent in E differ by exactly 1. Indeed, either v is a child of u or it is the parent of u. We say that an array in which any two adjacent elements differ by $+1$ or -1 satisfies the ± 1 *restriction*. Since L is such an array and we need to answer rmq-queries on L, we now concentrate on developing an efficient preprocessing method for such arrays.

Let $A[1, \ldots, n]$ be an array that satisfies the ± 1 restriction. We assume for the sake of simplicity that $\log n$ is an even integer. In this new approach we partition the array A into equal sized blocks of length $\left(\frac{1}{2}\right) \log n$. For each of these blocks we store the minimum elements in an additional array $A'[1, \ldots, \frac{2 \cdot n}{\log n}]$, see Figure 2.4. Furthermore, we need to keep track of the positions of the elements that are the minima in the blocks. We define an equal sized array B such that $B[i]$ is the position of the minimum element in block i. Recall that we need the positions of minimum elements to answer lca-queries. We preprocess A' for rmq-queries with the Sparse Table Algorithm described above. On A' the complexity is $O(n)$.

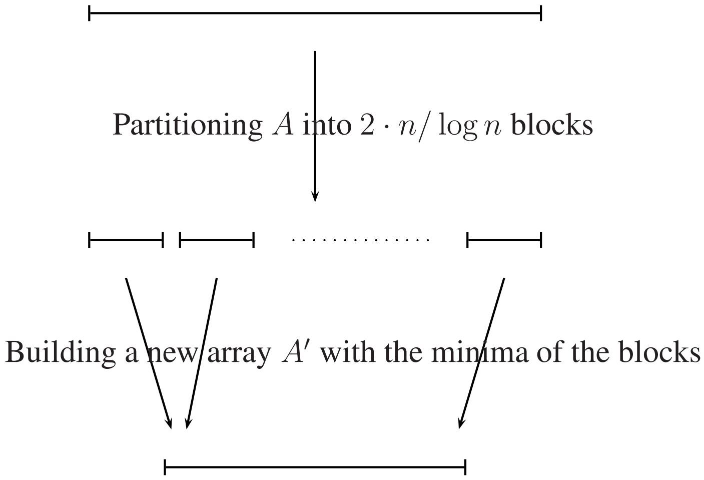


Figure 2.4: Partitioning the array A with the linear preprocessing solution.

We now consider rmq-queries on A. Let $1 \leq i < j \leq n$ be two integers so that i is in block h_i and j is in block h_j of the partitioned array A. Let $h_i < h_j$. The minimum of A in the range $[i, \ldots, j]$ is equal to the minimum of the following three values.

- The minimum in block h_i starting at position i to the end of the block
- The minimum in block h_j starting at the beginning of the block to position j.
- The minimum element in the subarray $A'[h_i + 1, h_j - 1]$. (∞ if $h_j = h_i + 1$)

To see this, recall the definition of the array A'. Obviously, we can compute the third minimum in constant time since we have preprocessed A' for such queries. However, it remains to develop an efficient method for the two *in-block* queries of the first two minima.

We now concentrate on these in-block queries. Obviously, we cannot afford to preprocess each block completely. However, this is the point where we use the ± 1-property of A.

We have $2n/\log n$ blocks each having length $1/2 \cdot \log n$. Nevertheless, we claim that we have only $O(\sqrt{n})$ "distinct" blocks. To see this, we normalize the blocks by subtracting the first element of each block from all other elements. Thereby we obtain $2n/\log n$ normalized blocks that have the following properties.

- The length is still $1/2 \cdot \log n$
- The first element in each block is zero
- The ± 1-property is still satisfied

We bound the number of normalized blocks having the above properties. Observe that such blocks can be encoded by a string of length $1/2 \cdot \log n - 1$ over the alphabet $\{+1, -1\}$. The number of such strings is given by

$$2^{1/2 \cdot \log n - 1} = (1/2) \cdot \sqrt{n} \tag{2.1}$$

Thus, we have

Lemma 2.29 (Number of Normalized Blocks). *In an array satisfying the ± 1-property there are $O(\sqrt{n})$ normalized blocks of length $1/2 \cdot \log n$*

We preprocess each normalized block by the naive algorithm. That is, we create $O(\sqrt{n})$ tables that contain the answers for all possible in-block queries. Obviously, the preprocessing time for each table is $O(\log^2 n)$ so that we have a total preprocessing time of $O(\sqrt{n}\log^2 n)$.

Additionally, we have to assign the appropriate normalized block to each block of the array A. This can be done by representing each block as an integer that corresponds to the binary representation of the encoding string (replacing $+1$ with 1 and -1 with 0). Obviously, this can be achieved in time $O(n)$, and assuming that the tables can be accessed according to these keys we have.

Theorem 2.30 (RMQ Solution for ± 1 Arrays). *Let $A[1, \ldots, n]$ be an array that satisfies the ± 1 property. Then we can preprocess A in time $O(n)$ such that rmq-queries can be answered in time $O(1)$.*

Corollary 2.31 (LCA Solution). *Let T be a rooted tree with n nodes. Then we can preprocess T in time $O(n)$ such that lca-queries can be answered in constant time.*

Proof. Theorem 2.30 and Lemma 2.27. □

Chapter 3

Approximate Pattern Matching With Error Trees

3.1 Algorithm Outline

Our method uses suffix trees as basic data structure. When searching for substrings of the text t that match a given pattern p with at most one mismatch, we match p exactly in the tree as far as possible. We then use information about substrings of t that match each other with one mismatch to obtain those substrings that match the pattern with at most one mismatch. This information is stored in a data structure that we call *error tree*, which is a simple extension to the suffix tree.

Throughout this chapter we denote the compact suffix tree for t with T.

3.2 Preliminary Definitions

We begin with defining *error trees* and *extension trees*. These trees are the core elements of our index structure. However, the information provided by extension trees is implicit in T. Despite of that we introduce extension trees here. We show later on how to use the implicit information in T to make them obsolete. The concept of extension trees is helpful in the development and understanding of the method.

Both error trees and extension trees are defined on internal nodes and the root node of the suffix tree T for a given text t. Additionally, we define extension trees on substrings of t.

Definition 3.1 (Error Tree of a Node). *Let v be an internal node or the root in T. Let $path(v) = \alpha$, where $|\alpha| = m$, $0 \leq m \leq n$. The error tree for node v, denoted $ET(v)$, is a Σ^+-tree built by those suffixes suf_j of t for which we have $suf_{j-m} = \beta suf_j$ and β matches α with at most one mismatch. The label of the leaf node that corresponds to suf_j in $ET(v)$ is $j - m$.*

On the one hand, an error tree of a node v serves the purpose of keeping track of all substrings of the text that match $path(v)$ with at most one mismatch. On the other hand, error trees provide a compact representation of the information needed to verify approximate occurrences of patterns.

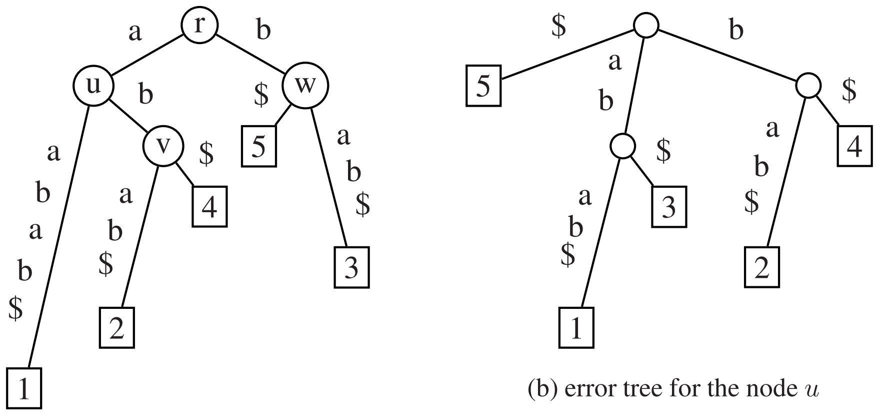
Note that by demanding that β matches α with at most one mismatch, we also cover the case $\beta = \alpha$.

We store the information about substrings that match with at most one mismatch in form of suffixes – similarly to suffix trees. Recall that every substring of t is a prefix of some suffix suf_i of t. Thus, if we have an external node with label i in the error tree of a node v, we know that a substring of t beginning at position i matches $path(v)$ with at most one mismatch. More specifically, if $|path(v)| = m$, $t(i, i+m-1)$ matches $path(v)$ with at most one mismatch.

On the other hand, for reasons given below, we need a compact representation of those parts of the suffixes that are not involved in the approximate match, more precisely the m-th suffix of all suffixes suf_i for which we have a leaf with label i in the error tree. The m-th suffix of a suffix suf_i is obviously also a suffix of t, namely suf_{i+m}. The reason for this is that an error tree for a node v is used to determine substrings $\beta\alpha$, where β matches $path(v)$ with one mismatch. Thus, we are interested in all suffixes suf_{i+m} that have a prefix α, and hence we build a tree of all suffixes suf_{i+m} but label the corresponding external nodes with i.

Error trees can also be considered in analogy to subtrees of suffix trees. Here, we find in a subtree rooted at node v, $|path(v)| = m$, an external node with label i and a path from v to this external node that has (path) label suf_{i+m}. In error trees we have, in contrast to subtrees of suffix trees, not only suf_{i+m} in the tree where $suf_i(1,m) = path(v)$, but additionally suf_{i+m} where $suf_i(1,m)$ matches $path(v)$ with one mismatch.

To prevent confusion, we say a suffix suf_i *appears* in an error tree if there is an external node with label i in the tree. We say a suffix suf_i *builds* an error tree if there is a path from the root of the error tree to an external node with (path) label suf_i.



(b) error tree for the node u

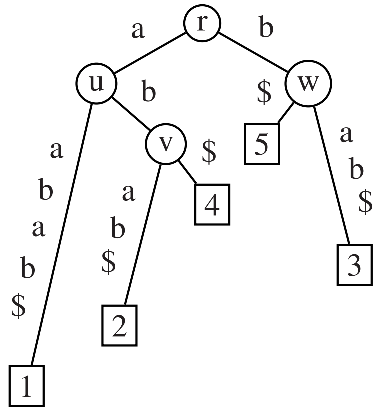
(a) suffix tree

Figure 3.1: Error tree of the node u of the suffix tree for the text $aabab\$$. Observe that both suf_5 and suf_3 appear in $ET(u)$ since their prefix of length $|path(u)|$ matches $path(u)$ with one mismatch.

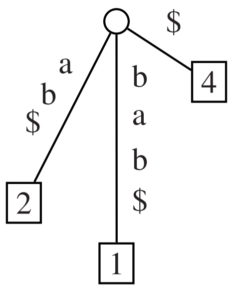
Corollary 3.2 (Error Tree of the Root Node). *Let u be the root of T. Then $ET(u) = T$.*

Proof. We have $path(u) = \epsilon$ and $suf_i = \epsilon suf_i$ for all suffixes suf_i of t. Thus, all suffixes suf_i build $ET(u)$ and appear in $ET(u)$ since $|path(u)| = 0$. □

Definition 3.3 (Extension Tree of a Node). *Let v be an internal node in T or the root of T and $path(v) = \alpha$, $|\alpha| = m$, $\alpha \in \Sigma^*$. The extension tree for v, denoted $XT(v)$, is a Σ^+-tree built by those suffixes suf_j of t for which we have $suf_{j-m-1} = \alpha a suf_j$, where $a \in \Sigma$ is an arbitrary character. The label of the leaf that corresponds to suf_j in $XT(v)$ is $j - m - 1$.*



(a) suffix tree



(b) Extension tree of node u

Figure 3.2: Extension tree of the node u of the suffix tree for the text $aabab\$$

Additionally, we define extension trees on substrings of t. This helps us in designing and analyzing the search algorithm in Section 3.3.

Definition 3.4 (Extension Tree of a Substring). *Let $\alpha \in words(T)$. Then the extension tree $XT(\alpha)$ is the Σ^+-tree that is built by those suffixes suf_j with $suf_{j-m-1} = \alpha a suf_j$ for any $a \in \Sigma$. Again, we set the label of the leaf node that corresponds to suf_j in $XT(\alpha)$ to $j - m - 1$.*

Note that if the path from the root of T along a string α, where $\alpha \in words(T)$, does not end at a node, $XT(\alpha)$ is a well defined subtree of T. There is a unique character $a \in \Sigma$ so that every substring α continues with a. Therefore, the suffixes suf_j building $XT(\alpha)$ are equal to the suffixes building $T_{\alpha a}$. Conversely, if α ends at a node, there are at least two different of such subtrees containing suffixes that build the extension tree.

3.3 Search Algorithm

Using the trees we have described in the previous section, we are able to locate all substrings of the text t that match a given pattern p with at most one mismatch in time $O(m + occ)$, where $|p| = m$ and occ is the number of such substrings.

The search algorithm matches a pattern p in the suffix tree exactly as far as possible. The last node u on the path from the $root$ to the point reached by the matching is used to access the appropriate error tree and extension tree. Throughout this section let $p(1, k)$, $k < m$, be the longest prefix of p that can be matched exactly in T. Let u be the deepest node on $path(p(1, k))$ and $path(u) = p(1, j)$, $j \leq k$. We demand that $k < m$ as we assume that there is no exact occurrence of p in t. This simplifies the analysis. However, we can easily handle exact matches as explained below.

In $ET(u)$ the algorithm looks for a path labeled $p(j+1, m)$, whereas it scans for $p(j+2, m)$ in $XT(u)$. The leaves in these subtrees correspond to suffixes having a prefix that matches p with one mismatch. Finally, if the exact matching of p ends inside an edge, i.e. $k \neq j$, one character $p(k+1)$ is skipped in the suffix tree for it is surely erroneous and $p(k+2, m)$ is scanned for in $XT(p(1, k))$, which corresponds to a subtree of T as mentioned above. This algorithm is quite simple and straightforward if we consider properties of substrings of t that match p with one mismatch. We do this formally when proving the correctness of the algorithm.

The description above is transcribed in algorithm *SearchAlgorithm*.

Algorithm 4 SearchAlgorithm

```
Match p in T as far as possible, let p(1, k) be the longest prefix of p that can be matched
Let u be the deepest node on path(p(1, k)) and path(u) = p(1, j)
Collect leaves of ET(u)_{p(j+1,m)}
Collect leaves of XT(u)_{p(j+2,m)}
if j ≠ k then
   // the case that p(1, k) does not end at a node
   Collect leaves of XT(p(1, k))_{p(k+2,m)}
end if
Return all collected leaves
```

3.3.1 Time Complexity of the Search Algorithm

Lemma 3.5 (Time Complexity of the Search Algorithm). *Algorithm SearchAlgorithm runs in time $O(m + occ)$, where occ is the number of occurrences of substrings in t that match p with at most one mismatch.*

Proof. Matching a pattern in a tree and collecting the corresponding leaves can be achieved in time $O(|pattern| + occ')$, where occ' is the number of leaves in the appropriate subtree, see Theorem 2.21. Here, all leaves that are returned correspond to substrings matching p with one mismatch. However, we have to verify that no leaf is returned several times. Obviously, in any error tree or extension tree, a certain leaf appears only once. Consider the trees in which returned leaves appear. Suppose a returned leaf, say suf_i, is returned twice and appears in $ET(u)_{p(j+1,m)}$ and $XT(u)_{p(j+2,m)}$. Since we assume that there are no exact matches, there must be a mismatch between $suf_i(1, m)$ and $p(1, m)$. However, by definition the possible mismatch locations of suffixes that appear in $ET(u)_{p(j+1,m)}$ are before node u, whereas the only possible mismatch location of a suffix that appears in $XT(u)_{p(j+2,m)}$ is directly after node u. This contradicts the fact that suf_i can appear in both trees because there is a mismatch, but the mismatch locations cannot agree. A similar argument shows that a suffix that appears in $XT(p(1,k))_{p(k+2,m)}$ can appear in neither $ET(u)_{p(j+1,m)}$ nor $XT(u)_{p(j+2,m)}$. In both trees, no suffixes appear with mismatch location $k+1$.

To see why the total search time in the trees is $O(m)$, we consider the following. We match a pattern in a tree at most four times and the pattern lengths are obviously $\leq m$ since only substrings of p are involved. The appropriate trees can be accessed in constant time assuming that we have pointers to $ET(u)$ and $XT(u)$ for any node $u \in T$. Therefore, we can bound the running time of the algorithm by $O(m + occ)$.

□

3.3.2 Correctness of the Search Algorithm

We start this section with two lemmas that characterize the possible mismatch locations between two substrings of t that match with one mismatch and between an arbitrary pattern and a substring of t that match with one mismatch. We say a pattern $p(1,m)$ and a substring $t(i, i+m-1)$ match with one mismatch and mismatch location $j+1$ if $p(1,j) = t(i, i+j-1)$, $p(j+2,m) = t(i+j+1, m-1)$, and $p(j+1) \neq t(i+j)$.

Lemma 3.6 (Mismatch Locations of Substring). *Let α be a substring of t. Let $\{v_0, v_1, \ldots, v_k\}$ be the nodes on the α-path in T and $\{i_0, i_1, \ldots, i_k\}$ be the string depths of $\{v_0, v_1, \ldots, v_k\}$. Let $t(j, j+|\alpha|-1) = \beta$ be a substring of t that matches α with one mismatch. Then the mismatch location between α and β is in the set $\{i_0+1, i_1+1, \ldots, i_k+1\}$.*

Proof. α and β can be written as $\alpha' a \alpha''$ and $\alpha' b \alpha''$ respectively, where characters a and b are distinct and mark the mismatch and $\alpha', \alpha'' \in \Sigma^*$. Thus, α' is a right branching word. As mentioned in Section 2, it then follows that there exists a node u with $path(u) = \alpha$ in T. □

To simplify the discussion, we say a mismatch between two arbitrary substrings α and β matching with one mismatch occurs after a node and mean that $\alpha(j'+1) \neq \beta(j'+1)$, where u' is an arbitrary node in T and $|path(u')| = j'$ so that the two substrings have $path(u')$ as longest common prefix. Similarly, we say a mismatch between a pattern and a substring α occurs after a node and mean that there is a node $u' \in T$ so that $|path(u')| = j'$, $\alpha(1,j') = p(1,j')$ and $\alpha(j'+1) \neq p(j'+1)$.

Lemma 3.7 (Mismatch Locations of Patterns). *Let p, $|p| = m$, be a pattern and t be a text. Let $p(1,k)$ be the longest prefix of p that can be matched exactly to any substring $t(i', i'+k-1)$ of t. Let $t(i, i+m-1)$ be a substring of t matching p with one mismatch. Then the mismatch location is either after some node on the path from the root to the endpoint of $path(p(1,k))$ or it is at position $k+1$.*

Proof. Suppose the mismatch location is at position l so that $l \leq k$. As $t(i, i+m-1)$ matches p with one mismatch, it follows that also $t(i, i+k-1)$ matches $p(1,k)$ with one mismatch. We note that both $t(i, i+k-1)$ and $p(1,k) = t(i', i'+k-1)$ are substrings of t. By Lemma 3.6

the mismatch location of these two substrings that match with one mismatch is after a node on $path(p(1,k))$.
Suppose now conversely that $l > k$. The mismatch location must be at position $k+1$ because both p and $t(i, i+m-1)$ match exactly up to the mismatch location, and $p(1,k)$ is the longest prefix of p that matches any substring of t exactly. □

Mismatches between substrings of t occur directly after nodes of T. When matching a pattern p to text t, we now know that all mismatches with mismatch location $l \leq k$ also occur after nodes and that there is only one additional mismatch location.

The following lemma connects substrings of t that match a pattern p with one mismatch and the error tree of u, the deepest node on $path(p(1,k))$.

Lemma 3.8. *The substring $t(i, i+m-1)$ matches p with one mismatch and mismatch location $\leq j$ if and only if suf_i appears in $ET(u)_{p(j+1,m)}$.*

Proof. Suppose that $t(i, i+m-1)$ matches p with one mismatch and mismatch location $\leq j$. We have to show that suf_i appears in $ET(u)$ and that $suf_i(j+1,m) = p(j+1,m)$.
If the substring $t(i, i+j-1)$ matches $p(1,j)$ with one mismatch, by Definition 3.1 suf_{i+j} builds $ET(u)$, and thus suf_i appears in $ET(u)$. As $t(i, i+m-1)$ matches p with one mismatch and the mismatch occurs between positions 1 and j, $t(i+j, i+m-1)$ matches $p(j+1,m)$ exactly. Thus, $suf_i(j+1,m) = p(j+1,m)$.
Suppose conversely that suf_i appears in $ET(u)_{p(j+1,m)}$. Then

$$suf_i(1,m) = suf_i(1,j)suf_i(j+1,m) = suf_i(1,j)p(j+1,m)$$

and $t(i, i+m-1)$ matches p with one mismatch since $suf_i(1,j)$ matches $p(1,j)$ with one mismatch by Definition 3.1. □

By Lemma 3.7 there are only two more mismatch locations that have to be considered, the mismatch location after node u, i.e. position $j+1$, and the mismatch location at position $k+1$. The next lemma covers position $j+1$.

Lemma 3.9. *The substring $t(i, i+m-1)$ matches p with one mismatch and mismatch location $j+1$ if and only if suf_i appears in $XT(u)_{p(j+2,m)}$.*

Proof. Suppose $t(i, i+m-1)$ matches p with one mismatch at mismatch location $j+1$. suf_i appears in $XT(u)$ since $suf_i(1, j+1) = path(u)x$, where x is an arbitrary character that mismatches $p(j+1)$. By Definition 3.3 suf_{i+j+1} builds $XT(u)$, and since $t(i, i+m-1)$ matches p with one mismatch, $suf_{i+j+1}(1, m-j-1) = suf_i(j+2,m) = p(j+2,m)$.
Conversely, suppose that suf_i appears in $XT(u)_{p(j+2,m)}$. It is then easy to verify that $t(i, i+m-1)$ matches p with one mismatch.

□

For the case that $path(u) \neq p(1,k)$, however, we have as additional possible mismatch location position $k+1$. We have no node at that point, thus every substring $p(1,k)$ in t continues with an unique character a, which is distinct from character $p(k+1)$. Let again $t(i, i+m-1)$ be

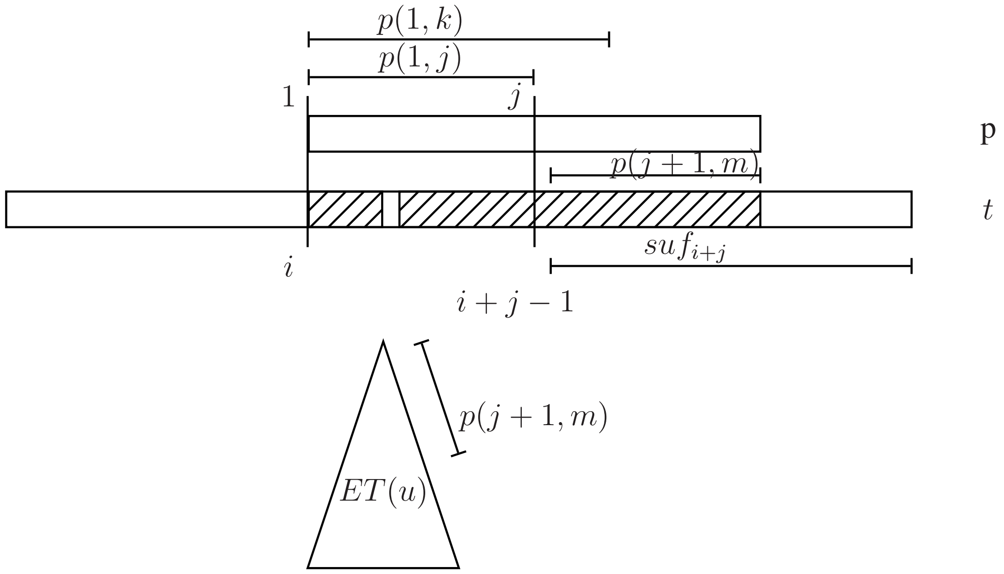


Figure 3.3: Strategy of the search algorithm for the case that the mismatch location is $\leq j$. Then suf_{i+j} builds $ET(u)$ and has a $p(j+1,m)$ prefix. The hatched areas of t are the parts that match the corresponding parts of p exactly.

a substring of t that matches p with one mismatch and mismatch location $k+1$. $t(i, i+m-1)$ has obviously a $p(1,k)a$-prefix, thus suf_i is a leaf in $T_{p(1,k)a}$. Furthermore, $t(i+k+1, m-1)$ matches $p(k+2,m)$ exactly so that we have a path $p(k+2,m)$ in $T_{p(1,k)a} = XT(p(1,k))$. Again, it is obvious that conversely every suffix appearing in $XT(p(1,k))_{p(k+2,m)}$ has a length m prefix that matches p with one mismatch. Thus, we have

Lemma 3.10. *Let $t(i, i+m-1)$ be a substring of t. $t(i, i+m-1)$ matches p with one mismatch if and only if suf_i appears in either $ET(u)_{p(j+1,m)}$ or $XT(u)_{p(j+2,m)}$ or $XT(p(1,k))_{p(k+2,m)}$.*

Theorem 3.11 (Search Algorithm). *Algorithm SearchAlgorithm correctly finds all substrings of t that match p with one mismatch in $O(m + occ)$ time.*

Proof. The correctness of the search algorithm follows from Lemma 3.10. The algorithm distinguishes the cases that the longest matching prefix α of p ends either at a node in T or not. In both cases it searches in $ET(u)$ so that all substrings matching with one mismatch and mismatch location $\leq j$ are found. Moreover, it searches in $XT(u)$ in both cases so that all substrings with mismatch location $j+1$ are discovered. For the case that α does not end at a node, it also searches for substrings with mismatch location $k+1$. No other mismatch locations are possible. The running time follows from Lemma 3.5. □

We have seen that using error trees and extension trees an efficient search for substrings of t that match a pattern p with one mismatch is possible. The worst case searching time of

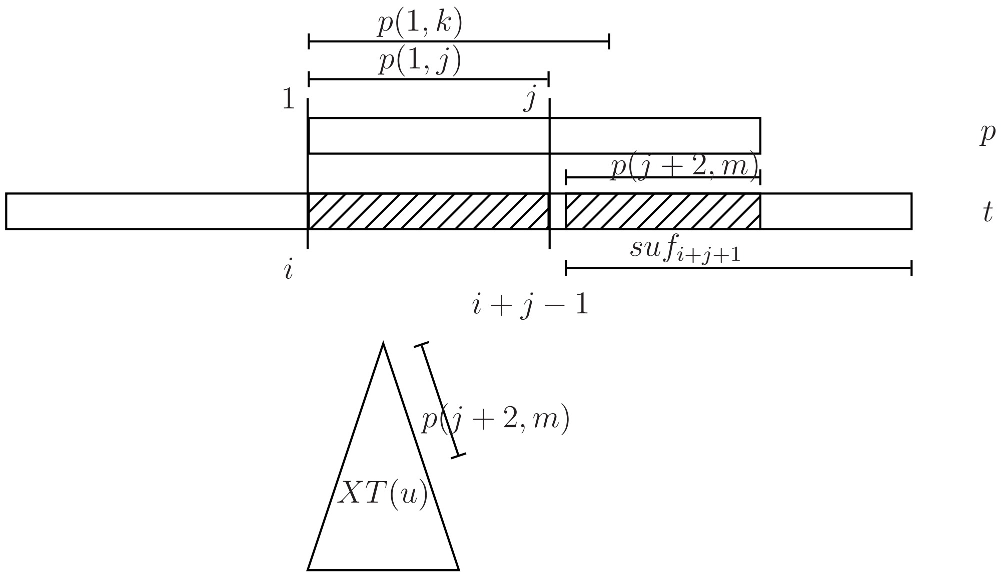


Figure 3.4: Strategy of the search algorithm for the case that the mismatch occurs at position $j+1$. suf_{i+j+1} builds then $XT(u)$ and has a $p(j+2, m)$ prefix. Again the hatched areas of t are the parts that match the corresponding parts of p exactly.

$O(m+occ)$ is asymptotically equivalent to the case of exact pattern matching. We have assumed that there are no exact occurrences of p in t to simplify analysis. Suppose now that there are such exact occurrences. Obviously, we have $p(1, k) = p(1, m) = p$ in that case. If $path(p)$ ends at a node u, we just have to return all leaves in $ET(u)$. Suppose now that $path(p)$ does not end at a node. In that case, all exact occurrences of p are returned twice since the corresponding leaves are in $ET(u)_{p(j+1,m)}$ as well as in $XT(u)_{p(j+2,m)}$ according to Definitions 3.1 and 3.3. This problem can be solved by using *implicit extension trees* instead of extension trees, see Remark 3.31. We refer to Section 3.4.3 for details.

In the following section we give an algorithm for building the index structure that is needed for the search algorithm and investigate its time and space complexity.

3.4 Preprocessing and Complexity of the Index Structure

3.4.1 Construction of Error Trees

Throughout this section let v be a node of T and let u be its parent. Let $|path(u)| = k$, $|path(v)| = m$ and $label(u, v) = \alpha$.

Our algorithm for building the error trees makes use of relationships between the error tree of a node v and the error tree and extension tree of its parent node u. These relationships are formalized in Lemma 3.12. We exploit this by merging certain subtrees of $ET(u)$ and $XT(u)$ to

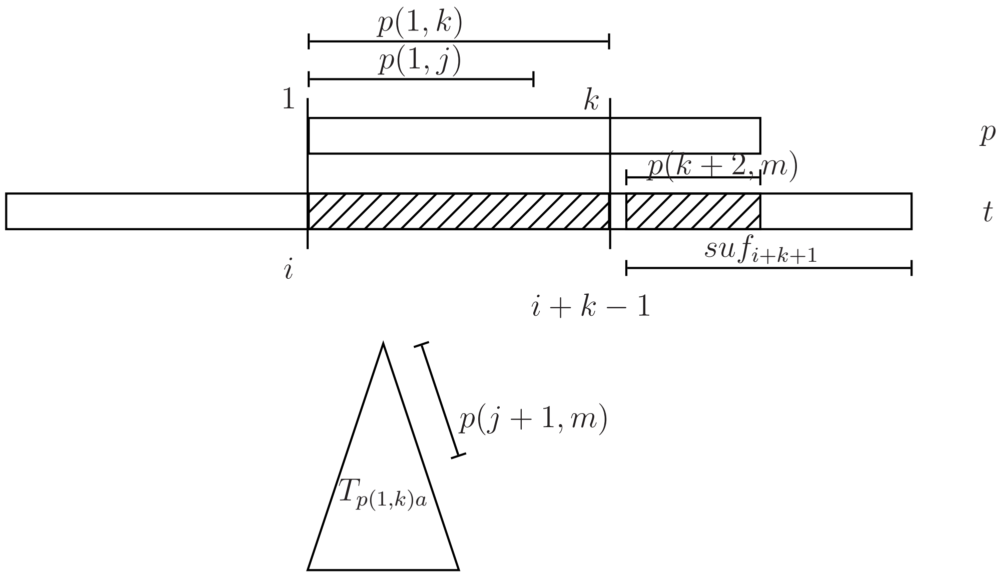


Figure 3.5: Strategy of the search algorithm for the case that the mismatch occurs at position $k+1$. suf_{i+k+1} builds then in $XT(p(1,k))$ and has a $p(k+2,m)$ prefix.

obtain $ET(v)$. The subproblem of merging two or more Σ^+-trees is handled in Section 3.5, and an efficient solution running in linear time with respect to the size of the resulting tree is given.

The intuition of our method for building the error trees and extension trees is the following. Recall Definitions 3.1 and 3.3. When we follow the edge from a node u to a child v, then the mismatch location in substrings that match $path(v)$ with one mismatch is either somewhere on the path from the $root$ to u, or it is directly after node u. It cannot be somewhere else on the edge (u,v) as mismatches between substrings of t occur after nodes (see Lemma 3.6). If the mismatch occurs on the path from the root to node u, we have already stored those suffixes that have a prefix which matches $path(u)$ with one mismatch in $ET(u)$. Obviously, if these prefixes also match $path(v)$ with one mismatch, they must be extendable by $label(u,v)$. On the other hand, if the mismatch occurs directly after node u, the substring surely must match the whole of $path(u)$ and then again $label(u,v)$ with the exception of its first character. By storing all the information at node u in trees, we can quickly find those suffixes that have a prefix that still matches $path(v)$ with one mismatch and thus successively construct the trees for deeper nodes. Note that the principles of this strategy are analogous to those of the search algorithm.

Lemma 3.12 (Suffixes Appearing in $ET(v)$). *A suffix suf_i appears in $ET(v)$ if and only if one of the following two conditions is satisfied:*

- *suf_i appears in $ET(u)_\alpha$*
- *suf_i appears in $XT(u)_{\alpha(2,|\alpha|)}$*

Proof. Let suf_i be a suffix that appears in $ET(v)$. Recall that by Lemma 3.6 mismatches between substrings occur after nodes. We now distinguish the cases that the mismatch between $path(v)$ and $suf_i(1, m)$ occurs after node u and all other possible mismatch locations (after nodes on the path from the root to u).

- **The mismatch occurs after some node on the path from the root to u, excluding u**
  We have to verify that suf_i appears in $ET(u)$ and that $suf_i(k+1, m) = \alpha$. Both is true since $suf_i(1, m)$ matches $path(v)$ with one mismatch and the mismatch occurs before position k.

- **The mismatch occurs after node u**
  The mismatch location is $k+1$. Again, we have to verify that suf_i appears in $XT(u)$ and that $suf_i(k+2, m) = \alpha(2, |\alpha|)$. This holds because $suf_i(1, k) = path(u)$, $suf_i(k+1) \neq \alpha(1)$, and thus $suf_i(k+2, m) = \alpha(2, |\alpha|)$ since $suf_i(1, m)$ matches $path(v)$ with one mismatch.

No other mismatch locations are possible, see Lemma 3.6.
Conversely, we have to show that every suffix suf_i that satisfies one of the two conditions appears in $ET(v)$. But this can be seen easily by verifying that in both cases $suf_i(1, m)$ matches $path(v)$ with one mismatch. Thus, the lemma is proven. □

Corollary 3.13 (Suffixes that Build ET(v)). *A suffix suf_j builds $ET(v)$ if and only if it builds either $ET(u)_\alpha$ or $XT(u)_{\alpha(2,l)}$*

Proof. This follows directly from Lemma 3.12. If suf_j builds $ET(v)$, suf_{j-m} appears in $ET(v)$. All suffixes suf_{j-m} that appear in $ET(v)$ appear in either $ET(u)_\alpha$ or $XT(u)_{\alpha(2,|\alpha|)}$. Recall that a suffix suf_{j-m} appears in an error tree if there is an external node with label $j-m$ in the tree and that a suffix suf_j builds an error tree if there is a path from the root to an external node with (path) label suf_j. If suf_i appears in $ET(u)$, we have that $suf_{j-|\alpha|}$ builds $ET(u)$ and, since that suf_j builds $ET(u)_\alpha$. Furthermore, if suf_i appears in $XT(u)$, $suf_{i-|\alpha|+1}$ builds $XT(u)$, and the claim holds for analogous arguments.
Conversely, it is again easy to see that suffixes that build $ET(u)_\alpha$ or $XT(u)_{\alpha(2,|\alpha|)}$ build $ET(v)$. □

Thus, we have seen how to obtain the suffixes that build $ET(v)$ by accessing appropriate subtrees of $ET(u)$ and $XT(u)$. The suffixes that build those two subtrees are exactly those suffixes that build $ET(v)$. It remains to consider the question of how to build the new error tree for v in an efficient way. Intuitively, this can be accomplished by simply merging the appropriate subtrees $ET(u)_\alpha$ and $XT(u)_{\alpha(2,|\alpha|)}$. For definition and solution of the tree merging problem, we again refer to Section 3.5.

We have all ingredients for the procedure that builds an error tree for a given internal node v.

Procedure 5 *BuildET(v)*

```
Let u be the parent node of v and label(u, v) = α
ET(v) ← MergeTrees(ET(u)_α, XT(u)_α(2,|α|))
```

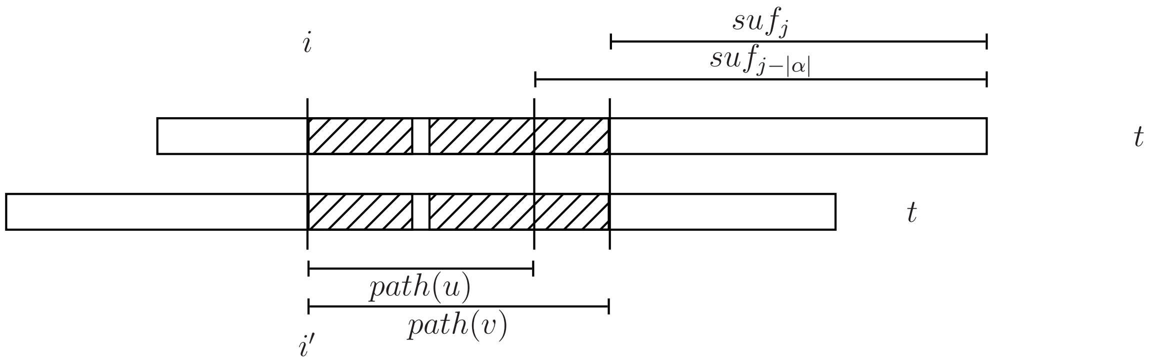


Figure 3.6: Illustration of case one of Lemma 3.12. If the mismatch occurs before node u, then suf_{j-1} builds $ET(u)$ and suf_j builds both $ET(u)_\alpha$ and $ET(v)$.

The correctness of this approach follows easily with the help of Lemma 3.33. This lemma states that the tree obtained by merging two trees is exactly the tree that is built by those strings that build one or both of the trees to be merged. Thus, we have the following useful lemma.

Lemma 3.14 (Error Tree by Merging). *The error tree for node v, $ET(v)$, is obtained by merging $ET(u)_\alpha$ and $XT(u)_{\alpha(2,|\alpha|)}$.*

Proof. This follows directly from Lemmas 3.12 and 3.33. □

Corollary 3.15. *Procedure BuildET correctly determines $ET(v)$.*

Proof. Lemma 3.14 □

The extension trees for a given node v are also obtained by merging subtrees. Unlike the case of building the error trees, the trees to be merged are subtrees of T and not subtrees of already determined error or extension trees. Recall that by Definition 3.3 a suffix suf_j of t builds $XT(v)$ if suf_{j-m-1} has a $path(v)a$ prefix, where $a \in \Sigma$ is an arbitrary character. Therefore, the leaves in the subtrees $T_{\alpha a}$, $\forall a \in \Sigma$, are exactly those suffixes that appear in $XT(v)$, and thus the labels of the paths from the root of $T_{\alpha a}$ to these leaves are the suffixes that build $XT(v)$. This discussion and Lemma 3.33 proof Lemma 3.16.

Lemma 3.16 (Correctness of BuildXT). *Procedure BuildXT correctly determines $XT(v)$.*

Algorithm *BuildTrees* visits the internal nodes and the root of T by a DFS and builds the error tree and extension tree for each node v. Procedures *BuildET* and *BuildXT* work as previously defined and can be used since the trees of the parent node u are already determined at the time v is visited. Recall that by Corollary 3.2, we have $ET(root) = T$.

Procedure 6 *BuildXT(v)*

$XT \leftarrow \emptyset$
for all $(v, w) \in T$ **do**
  Let x be the first character of $label(v, w)$
  $XT \leftarrow MergeTrees(XT, T_{path(v)x})$
end for
return XT

Algorithm 7 BuildTrees

Let $u \leftarrow root(T)$
Set $ET(u) \leftarrow T$
BuildXT(u)
Visit all internal nodes v of T by DFS
for all $v \in T$ **do**
  BuildET(v)
  BuildXT(v)
end for

3.4.2 Time and Space Complexity

Time and Space Complexity in Dependence of the Height of T

In the following we analyze the time and space complexity of the error trees and extension trees in dependence of the height h of the suffix tree T. Additionally, we derive worst case bounds that do not depend on the parameter h. The intention of this approach is that probabilistic analysis of suffix trees reveals that h converges to a value that is proportional to $\log n$. We then use both results to analyze the average case complexity of the trees in Section 3.6.

Lemma 3.17 (Edge Labels are Substrings of the Text). *Let (u, v) be an edge in an error tree or extension tree. Then $label(u, v)$ is a substring of t.*

Proof. Any error or extension tree is constructed by merging. Since all trees originate from T, we have by Definition 3.32 that all occurring edge labels are substrings of t. Recall that we have $words(T) = words(T') \cup words(T'')$ when merging two trees T' and T''. □

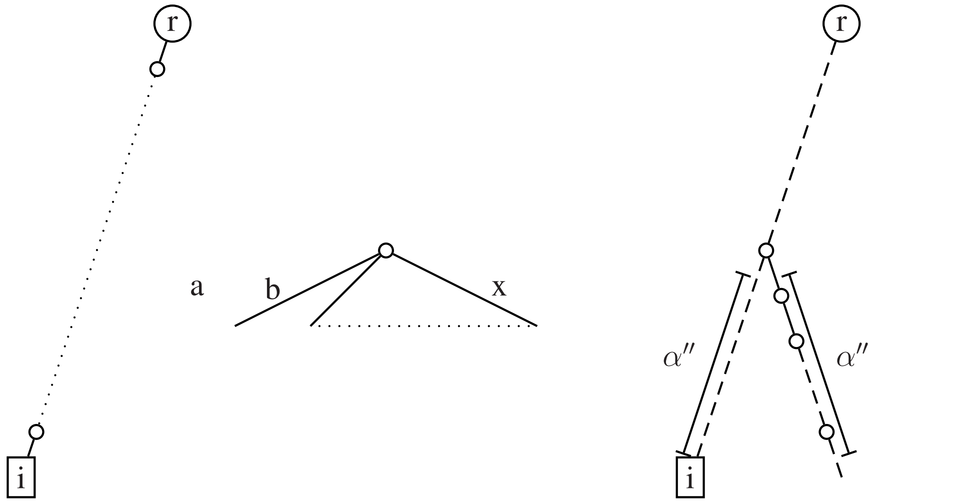
For the next lemma we use some notation that makes the considerations easier. Let β be a prefix of suf_i and α be a substring of t that matches β with one mismatch, where $\beta(j+1) = a \neq b = \alpha(j+1)$. Then we call the pair $(j+1, b)$ a specific mismatch of suf_i, i.e. a mismatch location together with a certain mismatch character.

Lemma 3.18 (Space Complexity of Error Trees). *The total size of all error trees is $O(n \cdot h^2)$*

Proof. We proof this by showing that the number of error trees in which a suffix suf_i can appear is bounded by $O(h^2)$.
If suf_i appears in $ET(u)$, by Definition 3.1 some prefix of suf_i matches $path(u)$ with at most

one mismatch. Let $path(u) = \alpha$ and let β be a prefix of suf_i such that α and β match with one mismatch. Note that all paths that correspond to prefixes of suf_i are on the simple path from the root to the external node with label i in T. Since α and β are substrings of t, the mismatch location between α and β is after some node on the path from the root of T to the endpoint of $path(\beta)$. In particular, the mismatch location is after some node on the path from the root to i. Thus, for all prefixes of suf_i that match some α with one mismatch we have in all at most h mismatch locations and thus $O(h)$ specific mismatches. Note that the maximal number of specific mismatches for a given mismatch location is constant, i.e. $|\Sigma| - 1$. Now consider the number of nodes in whose error trees suf_i can appear for a certain specific mismatch. All such nodes lie on the same simple path. Indeed, let $(j+1, b)$ be the specific mismatch and $suf_i(1, m)$ be the longest prefix of suf_i that matches a substring of t with one mismatch for this specific mismatch. Then all such nodes are on the path $suf_i(1, j-1)bsuf_i(j+1, m)$. On any simple path in T there are at most h nodes, so for any specific mismatch suf_i can appear in at most h error trees. Thus, suf_i can appear in $O(h^2)$ error trees.

Edge labels are substrings of t and they can be stored with constant space, so the size of an error tree is obviously bounded by the number of suffixes that build it. For every suffix suf_j that builds an error tree, suf_{j-m} appears in the error tree. Since we have n suffixes in T and every suffix can appear in at most $O(h^2)$ error trees, we can bound the total size of all error trees by $O(n \cdot h^2)$. $\square$



(a) at most h mismatch locations, i.e. nodes on the path from the root to i

(b) at most $|\Sigma| - 1$ specific mismatches at each mismatch location

(c) at most h nodes in whose error tree suf_i can appear for a specific mismatch, i.e. the path that is determined by α''

Figure 3.7: Illustration of the proof of Lemma 3.18

We continue with analyzing the space complexity of extension trees.

Lemma 3.19 (Space Complexity of Extension Trees). *The total size of all extension trees is* $O(n \cdot h)$

Proof. For any node $u \in T$, $XT(u)$ cannot have more leaves than T_u as the number of suffixes that build $XT(u)$ is bounded by the number of suffixes that have a $path(u)$-prefix. The intersection of leaves that are in T_u and T_v for two nodes u and v having the same node depth is empty as none of u and v can be a child of the other. Thus, the combined set of leaves in the subtrees rooted at all nodes that have a certain node depth k has size $\leq n$. Thus, we can derive that the total size of the extension trees of all nodes with a certain node depth is $O(n)$, and we have an overall size of $O(n \cdot h)$ □

We have shown that the total space complexity of error trees and extension trees is $O(n \cdot h^2)$. We continue with analyzing the running time of our algorithm for building the trees. We show that it is linear in the size of the resulting trees and thus $O(n \cdot h^2)$ for the error trees and $O(n \cdot h)$ for the extension trees. The proofs are easy considering the running times of the used procedures.

Lemma 3.20 (Preprocessing Time of Error Trees). *The total time needed for constructing all error trees is* $O(n \cdot h^2)$

Proof. Algorithm *MergeTrees(T',T")* is linear in the size of the resulting tree T, see Theorem 3.35. Additionally, before each call of *MergeTrees* we have to match the label of the last edge in the error tree and extension tree of the parent of the considered node. Let v be a node and u be its parent. Let $label(u, v) = a\alpha$. T_v is a substructure of $ET(u)$ and $T_{path(v)a}$ is a substructure of $XT(u)$. To see this, recall Definitions 3.1 and 3.3. Thus, locating the endpoints of $path(a\alpha)$ in $ET(u)$ and $path(\alpha)$ in $XT(u)$ takes time $O(h')$ where h' is the number of nodes on the paths $a\alpha$ and α respectively. We just have to locate the edge whose label starts with the appropriate character and can then jump to the next node. Since both $ET(u)$ and $XT(u)$ are built by suffixes of t, the number of nodes on any path in one of the trees is obviously $\leq h$. Thus, we can bound the total time needed for searching by $O(n \cdot h)$, i.e. complexity $O(h)$ at each node. It follows that the total time is dominated by the time needed by *MergeTrees*, which is linear in the size of all error trees, i.e. $O(n \cdot h^2)$ according to Lemma 3.18. To see this, recall that every error tree originates from one call of *Merge Trees*. □

Lemma 3.21 (Preprocessing Time of Extension Trees). *The total time needed for the construction of all extension trees is* $O(n \cdot h)$

Proof. Procedure *BuildXT* performs at most $|\Sigma|$ *MergeTrees* operations. None of the resulting trees has more leaves than the final extension tree, and therefore the size of all resulting trees is bounded by the size of the final tree. Thus, we can derive that the total time needed by the $\leq |\Sigma|$ *MergeTrees* operations is $O(n')$, where n' is the size of the final tree. The total merging cost for building all extension trees is hence $O(n \cdot h)$ according to Lemma 3.19. Again, we have to locate the appropriate subtrees before merging. But to achieve this, we only have to follow one character of each outgoing edge, which can obviously be done in constant time. □

Theorem 3.22 (Preprocessing Complexity of the Index Structure). *The total time needed for building error trees and extension trees for all nodes u in T is $O(n \cdot h^2)$.*

Proof. Lemmas 3.20 and 3.21 □

Unfortunately, h can be of order $O(n)$, which leads to a complexity of $O(n^3)$. However, we expect h to be of order $O(\log n)$ so that the trees have a complexity of $O(n \log^2 n)$. Nevertheless, we give an additional worst case analysis that does not depend on h.

Recall that by Definitions 3.1 and 3.3 all error trees and extension trees are built by some suffixes of t. Therefore, the size of every tree is $O(n)$. The total cost for the merging part during the construction of error or extension tree is linear in the total size of the trees and thus $O(n^2)$. Noting that the total cost for locating the subtrees is $O(n \cdot h)$ and thus $O(n^2)$, we have the following theorem.

Theorem 3.23. *The worst case time and space complexity of error trees and extension trees is $O(n^2)$.*

We use this result when analyzing the average case behavior of our trees in Section 3.6. Here, we continue with improving the space requirements of error trees.

3.4.3 Space Improvement

In this section we show how to improve the space requirements of the index structure to $O(n \cdot h)$. The basic idea to improve the bound is to store the error trees of all nodes in one single tree, which we call *aggregate error tree* (AET), instead of storing all trees separately. Moreover, we show in this section how to dispense with extension trees by using the implicit information of the suffix trees. We outline the revised approach.

At the beginning the aggregate error tree AET for t simply consists of the error tree for the root of T, $ET(root) = T$. Then we merge recursively into the appropriate subtrees of AET. Let u be the root and v be a child of u in T, where $label(u, v) = \alpha$. Let $ET(u)$ be the error tree of u. Obviously, there is a node w in $ET(u)$ with $path(w) = \alpha$. According to the results of Section 3.4 we know that $ET(v)$ is obtained by merging $ET(u)_\alpha$ and $XT(u)_{\alpha(2,|\alpha|)}$. Therefore, one can simply merge $XT(u)_{\alpha(2,|\alpha|)}$ to $ET(u)_\alpha$ without copying the resulting tree and creating $ET(v)$ explicitly. Certainly, $ET(u)$ is affected by merging something to one of its subtrees. Nevertheless, we show that the subtree of $ET(u)$ that is affected by this approach is never accessed by the search algorithm.

Lemma 3.24 (Independence of Error Trees). *Let u be a node in T and v be a child of u where $label(u, v) = \alpha$. Let p be a pattern and $p(1, k)$ be the longest prefix of p that matches a substring of t exactly. Let u be the deepest node on $path(p(1, k))$ in T. Then $ET(u)_\alpha$ is never accessed by algorithm SearchAlgorithm.*

Proof. Recall that the search algorithm we gave in Section 3.3 matches $p(j + 1, m)$ in $ET(u)$, where $j = |path(u)| = |p(1, j)|$. Suppose that $p(j + 1, m) = \alpha\gamma$ for some $\gamma \in \Sigma^*$. Otherwise, $ET(u)_\alpha$ is surely not accessed. But there is an edge (u, v) that is labeled with α by assumption, so $p(j+1, j+|\alpha|)$ could be matched up to node v in T. However, this is a contradiction to the assumption that u is the deepest node on the $path(p(1,k))$. □

Corollary 3.25. *Let v be a successor of u in T and $path(u,v) = \alpha$. Then $ET(u)_\alpha$ is never accessed by algorithm SearchAlgorithm.*

Proof. This follows from Lemma 3.24. Since v is a successor of u in T, $ET(u)_\alpha$ is a subtree of $ET(u)_{label(u,w)}$ for some node w that is a child of u in T. □

Recall that T_u is a sub-structure of $ET(u)$ for all nodes $u \in T$. However, if there is a node v in T_u so that $path(u,v) = \alpha$, we have verified that $ET(u)_\alpha$ is not important for the search algorithm as it is not accessed. Thus, merging additional trees to these subtrees $ET(u)_\alpha$ of $ET(u)$ does not affect the functionality of $ET(u)$ for the search algorithm. This is the key observation that allows us to store the information provided by all error trees of nodes in one aggregate error tree.

Algorithm 8 BuildAET

$AET \leftarrow T$
Visit all internal nodes $v \in T$ by DFS
for all $v \in T$ **do**
  Let u be the parent of v in T, $\alpha = label(u,v)$, and $XT = XT(u)$
  MergeTrees$(AET_{path(v)}, XT_{\alpha(2,|\alpha|)})$
  BuildXT(v)
end for

Our algorithm for building the aggregate error tree AET for a text t and the corresponding suffix tree T works as follows. First of all, we set $AET \leftarrow T$, which corresponds to $AET \leftarrow ET(root)$ by Corollary 3.2. Note that we are copying T to AET. We visit all nodes $v \in T$ by DFS. For each node v let u be its parent and let $label(u,v) = \alpha$. We have already determined $XT(u)$ and simply merge $XT(u)_{\alpha(2,|\alpha|)}$ to $AET_{path(v)}$. Here, $path(v)$ corresponds to $path(v)$ in T. Moreover, with v we store a pointer to the root of $AET_{path(v)}$ as this is the subtree of AET that is used for searching in the case that $ET(v)$ is the appropriate error tree. By Corollary 3.25, this approach is correct and returns the same results as if we used $ET(v)$ instead of $AET_{path(v)}$ if we can verify that $AET_{path(v)} = ET(v)$ directly after merging $XT(v)$ to $AET_{path(v)}$. In that case, all further merging only affects subtrees of $AET_{path(v)}$ that are not relevant.

Lemma 3.26. *Let v be a node in T. Then $AET_{path(v} = ET(v)$ directly after v has been visited by algorithm BuildAET.*

Proof. We show this by induction over the node depth of v in T_u.

- **Induction Start**: v is the root of T.
  We set $AET = T$ when visiting v and $ET(v) = T$ by Corollary 3.2.
- **Induction Step**: We assume that the claim holds for all nodes with node depth $< k$.
  Let v be a node with node depth k. Let u be the parent of v. Since the node depth of

u is $k-1 < k$, by assumption $AET_{path(u)} = ET(u)$ directly after u was visited. Let $label(u, v) = \alpha$. Then $AET_{path(v)} = AET_{path(u)\alpha}$. As the nodes are visited by DFS, all nodes that are possibly visited between the visits of u and v are direct children of u. Let $v' \neq v$ be a child of u and $label(u, v') = \beta$. It can be seen easily that we can merge anything to $AET_{path(u)\beta}$ without altering $AET_{path(u)\alpha}$. Thus, $AET_{path(u)\alpha}$ is not affected by merging operations that take place before v is visited. It follows that $AET_{path(v)} = ET(u)_\alpha$ at first, and by Lemma 3.14, $AET_{path(v)} = ET(v)$ at the time the visit of v is finished.

□

Corollary 3.27. *Let (u, v) be an edge in AET at any point of its construction. Then $label(u, v)$ is a substring of t.*

Proof. This is a simple consequence of Lemmas 3.17 and 3.26. □

We now show that the space complexity of AET is only $O(n \cdot h)$.

Theorem 3.28 (Space Complexity of Aggregate Error Trees). *Let t be a text. The aggregate error tree AET for t can be constructed in time $O(n \cdot h^2)$ and uses $O(n \cdot h)$ space.*

Proof. Algorithm *BuildAET* starts with fixing $AET = T$. The size of this tree is obviously bounded by $O(n)$ and so is the time for constructing it, i.e. making a copy of T. Then, for all nodes $v \in T$, $XT(u)_{\alpha(2,|\alpha|)}$ is merged to $AET_{path(v)}$. The total size of all extension trees is $O(n \cdot h)$, see Lemma 3.19. For any node w a subtree of $XT(w)$ can be merged only a constant number of times (i.e. for each of its children) to AET. Thus, in all $O(n \cdot h)$ external nodes are added to AET, and hence we have a total size of $O(n \cdot h)$. If we recall the proof of Lemma 3.20, this can also be seen by noting that suf_i appears only once in AET for each specific mismatch. The time needed for merging the extension trees to the appropriate subtrees is obviously bounded by the time needed in the case in which the error trees are constructed solitarily, which is $O(n \cdot h^2)$. To see this, observe that at the time $XT(u)_{\alpha(2,|\alpha|)}$ is merged to $AET_{path(v)}$ both trees correspond to the trees that are merged by procedure *BuildET*(v). Besides, the time needed to navigate in AET for accessing the appropriate subtrees is the same as for navigating in the error trees of the single nodes, which is $O(n \cdot h)$. Thus, we have a total preprocessing time of $O(n \cdot h^2)$. □

As mentioned above, it is possible to dispense with extension trees both in the search algorithm and the algorithms for constructing error trees without changing the asymptotics of the whole method. Without going into detail, we show how this can be done by using *implicit extension trees* instead of the explicitly constructed extension trees of nodes. The intuition of this is quite easy, but the particular derivation is technical and reveals no interesting new aspects of the problem.

Definition 3.29 (Implicit Extension Trees for a Node). *Let v be a node in T and $path(v) = \alpha$. The implicit extension trees for v are $T_{\alpha a}, \forall a \in \Sigma$ so that there exists an outgoing edge of v whose label starts with character a.*

By Lemma 3.16 $XT(v)$ is obtained by merging the implicit extension trees of v. Thus, the leaves in the implicit extension trees correspond to leaves in $XT(v)$.

Lemma 3.30 (Implicit Extension Trees for Searching). *By using implicit extension trees instead of the extension tree of a node for searching, the asymptotic running time is not affected and the correctness is maintained.*

Proof. Consider algorithm *SearchAlgorithm*. The only line that is affected by this alteration is:

Collect leaves of $XT(u)_{p(j+1,m)}$

Instead of matching $p(j+1, m)$ in $XT(u)$, we match $p(j+1, m)$ in all implicit extension trees of u. There are at most $|\Sigma|$ implicit extension trees of u. Searching a pattern of length m in $|\Sigma|$ trees is asymptotically equivalent to searching it in only one tree as $|\Sigma|$ is a constant. The correctness of this approach is straightforward and follows from Definitions 3.3 and 3.29. □

Remark 3.31. *Using implicit extension trees instead of explicit extension trees allows us to deal with exact occurrences of the pattern in the text. As mentioned above, we have the problem that any exact occurrence of p in t appears in subtrees of both $ET(u)$ and $XT(u)$ whose leaves are returned. However, if using implicit extension trees, we can simply ignore the implicit extension tree that contains the exact occurrences, i.e. $T_{\alpha a}$, where $p(j+1) = a$. Thus, every exact occurrence is only returned once – within the leaves of $ET(u)_{p(j+1,m)}$.*

Similarly, the construction of the aggregate error tree AET can be achieved by using the implicit extension trees instead of the merged extension tree. We outline the approach.

Let again v be a node that is visited during the construction of an AET and let u be its parent. Let $path(u) = \alpha$ and $label(u, v) = \beta$. Then the suffixes appearing in $XT(u)_{\beta(2,|\beta|)}$ are obviously suffixes that have a prefix of the form $\alpha a\beta(2, |\beta|)$, where a is an arbitrary character (see Definition 3.3). It is easy to see that all these suffixes are leaves in some subtree $T_{\alpha a\beta(2,|\beta|)}$. It is also obvious that merging $T_{\alpha a\beta(2,|\beta|)}$ for all characters $a \in \Sigma$ to $AET_{path(v)}$ yields the same result as merging $XT(u)_{\beta(2,|\beta|)}$ to $AET_{path(v)}$.

In general, the tradeoff between explicit and implicit extension trees looks like that. Either we merge $O(|\Sigma|)$ implicit extension trees beforehand so that we have to match a given pattern in only one compacted version of all implicit extension trees, or we do not merge trees – then we have to match the pattern $O(|\Sigma|)$ times in each implicit extension tree. This is true both for the construction of error trees and for the search algorithm. At any rate, since $|\Sigma|$ is a constant, the asymptotic time complexity remains unaltered.

3.5 Merging Algorithm

3.5.1 Merging two Trees with a Coupled DFS Traversal

A subproblem arising in our approach for building the index structure is that of merging two or more Σ^+-trees into one. We present a simple algorithm that follows basically a coupled *DFS* traversal of both trees and that runs in linear time with repsext to the size of the resulting tree. Kim, Sim, and Park [KSP99] and Farach [Far97] have both given linear-time algorithms for merging two trees T_e and T_o in the course of presenting algorithms for constructing suffix trees for integer alphabets. Here, T_e denotes the suffix tree built by the set of even positions in the text (i.e. suf_j, where j is an even number), and T_o is the suffix tree built by the set of odd positions. The final suffix tree T is obtained by merging T_o and T_e. They point out that the main problem in merging those trees is to determine the length of the *longest common prefix* of two edge labels that start with the same character. In their case, this problem cannot be solved in constant time so that they need strategies for overcoming this problem in a post-processing step. In our case, however, we show that we can determine the longest common prefix in constant time, which leads to a much simpler and clearer algorithm.

We define the tree merging problem.

Definition 3.32 (Σ^+-trees merging problem). *Let T' and T'' be two compact Σ^+-trees. In the Σ^+-trees merging problem, one is interested in finding the tree T with the following properties:*

- *T is a compact Σ^+-tree*
- $words(T') \cup words(T'') = words(T)$

This definition meets the requirements of the intuitive idea about merging two trees. All information that is provided by both trees is encoded in the merged tree, and $words(T)$ can be regarded as the information provided by a Σ^+-tree T.

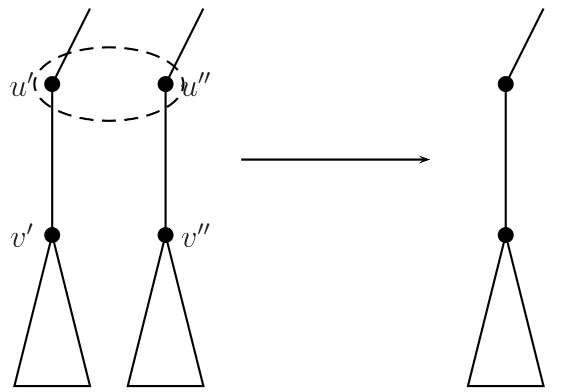
We start our development of the merging algorithm by considering an algorithm for merging two tries. The obvious procedure for accomplishing this task is a coupled *DFS* traversal of both tries. We start at the roots of both tries and then identify simultaneously edges of both tries that have edge label $a \in \Sigma$. Such edges can be merged, and we recursively merge the two subtrees. We do this for each character $x \in \Sigma$ to obtain the merged tree. At any node, if only one of the two subtrees has an outgoing edge that is labeled with character x, the recursion can be aborted for the subtree after that edge since there is no further merging possible. For merging two atomic tries this would be the approach of choice and its running time is clearly linear in the size of the resulting merged tree.

However, merging two compact trees is more challenging. For the sake of simplification consider just the first step of the procedure described above. We take two edges of both roots whose edge labels start with the same character, say a. As the edge labels represent strings of size $O(n)$, we surely cannot afford to compare the whole of the labels in order to verify equivalence. Another problem arises from the fact that we can only merge edges whose labels have the same length. So when encountering two labels where the shorter one is a prefix of the other one, we have to split the longer label at an appropriate length. This fact as well as some other details have to be considered when developing an algorithm for merging two compacted trees.

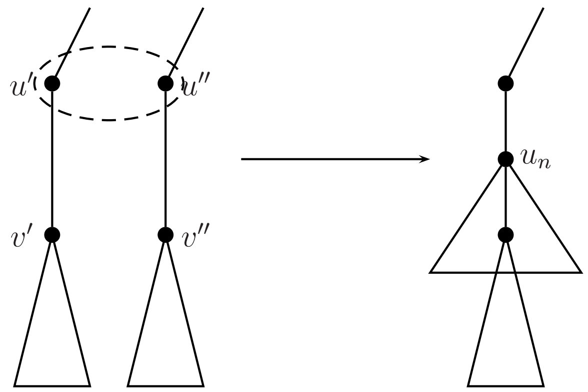
Throughout this section let (u', v') and (u'', v'') be two edges that we want to merge. Let $label(u', v') = \alpha'$ and $label(u'', v'') = \alpha''$, where $|\alpha'| = m'$ and $|\alpha''| = m''$. Consequently, we demand that α' and α'' start with the same character. We assume without loss of generality that $m' \leq m''$. Let $lcp(\alpha', \alpha'')$ be the length of the longest common prefix of α' and α''. In what follows we suppose that we can determine $lcp(\alpha', \alpha'')$ in constant time. We show how this can be done in Section 3.5.2. Before proceeding, we recall that by Corollary 3.27 all edge labels in an aggregate error tree are substrings of t.

We turn back to the problem of merging two compact trees. Recall the method for merging two tries described above and consider again the edges (u', v') and (u'', v''). Let $lcp(\alpha', \alpha'') = j$. We distinguish three cases.

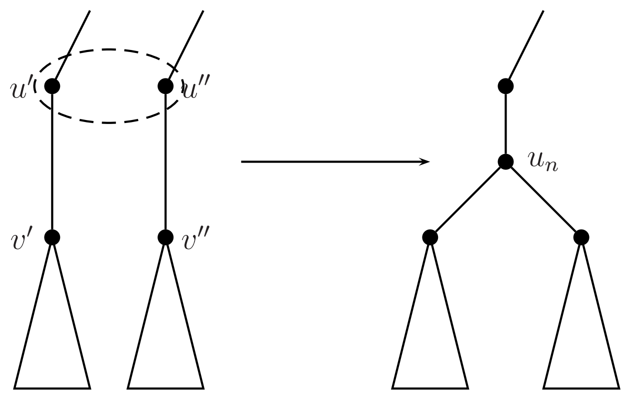
1. $\alpha' = \alpha''$
   In that case we can obviously merge the two edges and proceed with merging $T_{v'}$ and $T_{v''}$.



2. $lcp(\alpha', \alpha'') = j = m'$ and $m' < m''$, that is α' is a proper prefix of α''
   We break (u'', v'') and introduce a new node u''_n. We set $label(u'', u''_n) = \alpha''(1, j)$ and $label(u''_n, v'') = \alpha''(j + 1, m'')$. We merge (u', v') and (u'', u''_n) and proceed with merging $T_{v'}$ and $T_{u''_n}$.



3. $lcp(\alpha', \alpha'') = j$ and $j < m'$
   In that case, we break α' and α'' after position j and introduce new nodes u'_n and u''_n. We set $label(u', u'_n) = \alpha'(1, j)$ and $label(u'_n, v') = \alpha'(j + 1, m')$. Analogously, we set $label(u'', u''_n) = \alpha''(1, j)$ and $label(u''_n, v'') = \alpha''(j + 1, m'')$. We merge (u', u'_n) and (u'', u''_n) and have again created a situation that allows the application of the simple base case. Note, however, that in this case no further merging has to be done as $\alpha'(j + 1) \neq \alpha''(j + 1)$ and u'_n and u''_n have only one outgoing edge.



The above is already an informal description of our merging algorithm and it becomes clear that a solution to the longest common prefix problem yields this clean and simple merging algorithm that runs in linear time. We now formally specify the algorithm and prove its correctness.

Algorithm 9 MergeTrees(T', T'')

$u' \leftarrow root(T')$, $u'' \leftarrow root(T'')$
Visit(u', u'')
return T

To proof the correctness of the algorithm, we first show that there is an alternative formulation of the tree merging problem, equivalent to Definition 3.32. Instead of demanding that the combination of words in both trees and the words in the merged tree must correspond, we show that we can also build a merged tree of all strings that correspond to leaf paths in the original trees. (Here, a leaf path is any path from the root of a tree to an external node.) This is also an obvious fact as any word in a tree is a substring of a string that corresponds to a leaf path, compare Definition 2.4.

Lemma 3.33 (Tree Merging). *Let T' and T'' be two Σ^+-trees. The Σ^+-tree that is obtained by merging T' and T'' as defined in 3.32 is the Σ^+-tree that is built by those strings that label some path from the root to an external node in either T' or T''.*

Proof. According to Definition 3.32 we just have to show that $words(T) = words(T') \cup words(T'')$.
Let α be a word in T' or T''. Assume without loss of generality that $\alpha \in words(T')$. By Definition 2.4 there exists a node $u' \in T'$ with $path(u') = \alpha\beta$ for some $\beta \in \Sigma^*$. Moreover, u' is either a leaf node or it lies on the path to some leaf node v'. Let $path(v') = \alpha\beta\gamma, \gamma \in \Sigma^*$. If T is the Σ^+-tree that is built by the strings that correspond to leaf paths in T' or T'', there exists a node w in T that is either a leaf node or an internal node with $path(w) = path(v') = \alpha\beta\gamma$. Therefore, α is a word in T.
Suppose conversely that α is a word in T. Consequently, there is a node w in T with $path(w) = \alpha\beta$ for some β. This node is again either a leaf node or on the path to some leaf node w' in T, where $path(w') = \alpha\beta\gamma$ and $\gamma \in \Sigma^*$. As T is built by strings that correspond to leaf paths in

```
Visit(u′,u″)
  Merge u′ and u″
  for all a ∈ Σ do
    if exist edges (u′, v′) and (u″, v″) with labels α′ and α″ such that α′(1) = α″(1)=a then
      j ← lcp(α′, α″)
      if j < |α′| then
        Introduce new node u′_n after character j on edge (u′, v′). Set label(u′, u′_n) = α′(1, j)
        and label(u′_n, v′) = α′(j + 1, |α′|).
      else
        u′_n = v′
      end if
      if j < |α″| then
        Introduce new node u″_n after character j on edge (u″, v″). Set label(u″, u″_n) = α″(1, j)
        and label(u″_n, v″) = α″(j + 1, |α″|).
      else
        u″_n = v″
      end if
      Merge edges (u′, u′_n) and (u″, u″_n).
      Visit(u″_n, u″_n)
    end if
  end for
```

either T' or T'', there exists a leaf node u' in T' or T'' with $path(u') = \alpha\beta\gamma$, and therefore any word in T is either a word in T' or in T''. □

This alternative formulation of the tree merging problem allows us to give an easy proof of the correctness of our tree merging algorithm.

Theorem 3.34 (Correctness of MergeTrees). *Algorithm MergeTrees correctly merges two Σ^+-trees.*

Proof. To show that the tree T resulting from the algorithm is the correct merged tree, we have to show that T is a Σ^+-tree and that it is built by exactly those strings that correspond to leaf paths in T' and T'', according to Lemma 3.33.

- T is a Σ^+-tree
  It suffices to show that there is no node in T that has more than one outgoing edge whose label starts with a certain character since all other properties of Σ^+-trees are obviously satisfied. The proof is inductive. We notice that at the beginning of the merging no node violates the claim since nothing has been merged. Furthermore, two nodes u' and u'' are merged if and only if *Visit*(u', u'') is called. However, outgoing edges of u' and u'' that start with the same character are always merged up to the point where they share the longest common prefix. Thus, no merged node violates the claim. Additionally, we have to verify

that this holds for the newly created nodes. But this is also the case since for every newly created node u'_n, *Visit*(u'_n, u''_n) is called with some node u''_n.

- T is built by exactly those strings that are leaf paths in T' or T''.
  To show this, we again use the recursive nature of our algorithm. Suppose that *Visit*(u', u'') is called for two nodes u' and u'' that have no outgoing edges that start with the same character. Obviously, since the nodes are merged, the result of this call is a tree that is built by those strings that correspond to leaf paths in $T'_{u'}$ and $T''_{u''}$.
  Let now u' and u'' be two arbitrary nodes that are to be merged and suppose that the claim holds for all subtrees of u' and u''. Then applying one of the three rules for merging any two edges that start with the same character establishes the claim for u' and u'', and the claim follows by induction. Note that after merging (u', v') and (u'', v'') we still have paths from the merged node to nodes that correspond to v' and v'' that have labels α' and α'' and that by assumption the claim is true for $T'_{v'}$ and $T''_{v''}$.

□

We continue with analyzing the running time of the merging algorithm.

Theorem 3.35. *The running time of the algorithm is $O(n')$, where n' is the size of the resulting tree T, if we can determine the longest common prefix of two edge labels in constant time.*

Proof. The proof is straightforward. A node of T is constructed by each call of *Visit*. The work needed for each call of $Visit(u', u'')$ is constant as we assume that the alphabet is of constant size and that we can determine $lcp(\alpha', \alpha'')$ in constant time. Furthermore, we can determine the parts of the edge labels that can be merged in constant time since we know $lcp(\alpha', \alpha'')$ and an edge label is represented by a pair of indices (i, j). □

3.5.2 Finding the Longest Common Prefix of two Edge Labels

We have seen that our algorithm for merging two trees runs in linear time given a means of determining the length of the longest common prefix of two labels in constant time. In this section we show how to achieve this by performing lca-operations on the suffix tree T. Such queries can be performed in constant time with linear preprocessing, see Section 2.6. The following easy lemma is essential to our method.

Lemma 3.36 (LCP by LCA). *Let w' and w'' be two nodes in T so that α' is a prefix of $path(w')$ and α'' is a prefix of $path(w'')$. Let j be the string depth of $lca(w', w'')$. Then*

$$lcp(\alpha', \alpha'') = \begin{cases} m' & \text{if } m' \leq j \\ j & \text{otherwise} \end{cases}$$

Proof. We have that $lcp(path(w'), path(w'')) = j$, which follows from properties of Σ^+-trees. Since we assume that $m' \leq m''$, $lcp(\alpha', \alpha'') \leq m'$. Thus, the claim follows easily considering that α' and α'' are prefixes of $path(w')$ and $path(w'')$. □

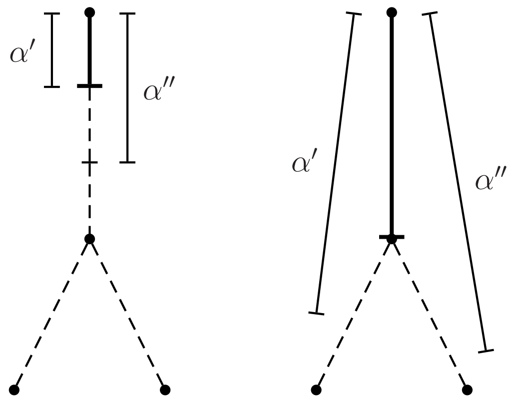


Figure 3.8: Illustration of the determination of $lcp(\alpha', \alpha'')$ with the help of answering an lca-query on two nodes w' and w''. The solid line corresponds to the longest common prefix.

The method we present here is very simple. It makes use of the fact that each edge label in an error, aggregate error, or extension tree is a substring of t, see Corollary 3.27. Recall that the labels α' and α'' of two edges we intend to merge are represented by a pair of indices (i', j') and (i'', j''). Let i' and i'' be the external nodes that correspond to suffixes $suf_{i'}$ and $suf_{i''}$. We show how to determine $lcp(\alpha', \alpha'')$ by querying $lca(i', i'')$.

Lemma 3.37 (LCP Solution with External Nodes). *Let $\alpha' = t(i', j')$ and $\alpha'' = t(i'', j'')$. Let i' and i'' be the external nodes with labels i' and i''. Let the string depth of $lca(i', i'')$ be j. Then*

$$lcp(\alpha', \alpha'') = \begin{cases} m' & \text{if } m' < j \\ j & \text{otherwise} \end{cases}$$

Proof. This follows directly from Lemma 3.36 since α' and α'' are prefixes of $suf_{i'}$ and $suf_{i''}$. □

The above lemma directly implies the method for determining $lcp(\alpha', \alpha'')$. Locate external nodes i' and i'', query $lca(i', i'')$, and derive $lcp(\alpha', \alpha'')$ from that. Since $lca(i', i'')$ can be answered in constant time (see Corollary 2.31) and the answer for $lcp(\alpha', alpha'')$ can be derived in constant time assuming that we have stored the string depths of all nodes, it only remains to consider how we can locate the external nodes i' and i''.

The solution is easy. We construct an array of size n where the i-th array entry holds a pointer to the external node with label i. This array can be filled in time $O(n)$ by simply performing a DFS on T or during the construction of T. Thus, we have the following theorem.

Lemma 3.38 (LCP Solution). *With linear preprocessing with respect to the text size, it is possible to find the length of the longest common prefix of two edge labels that have to be merged during the construction of error trees, extension trees, or aggregate error trees in constant time.*

Proof. Storing the string depths of all nodes needs $O(n)$ time according to Lemma 2.9. The array with pointers to external nodes can be constructed in linear time, compare the above discussion. The preprocessing time for the lca-queries is also $O(n)$, see Corollary 2.31. Thus, we need only linear preprocessing and can determine $lcp(\alpha', \alpha'')$ in constant time, which follows from Lemma 3.37 and the above discussion. □

For the sake of completeness we briefly summarize another method for determining the length of the longest common prefix in constant time, which we have developed in a first approach before the method described above. The basic idea of this second method is to store at each node v in an error or extension tree a pointer to a corresponding node w in the suffix tree, where $path(w) = label(u, v)$ and u is the parent of v. Having created such pointers, we obviously can use the nodes w to answer lcp-queries with lca-answers, see Lemma 3.36. Unfortunately, to be able to maintain and update those links in constant time during the construction of the trees, we have to augment the suffix tree with additional links of size $O(n \cdot h)$. Thus, the method described above is superior in both terms of simplicity and space complexity, although the size of the additional links of the second method would not affect the overall asymptotic space complexity of the index structure.

3.6 Asymptotic Behavior of Suffix Trees

We have shown that the complexity of the index structure depends on the height of the suffix tree h. Thus, we consider the asymptotic behavior of this parameter under a certain probabilistic framework. The results on which we rely are mainly due to Szpankowski, who released a series of papers on this topic, see [Szp00, Szp93] and the citations therein. We do not go into detail but present the analysis of the height of suffix tries in the case that the underlying text is created by a *memoryless source*. Obviously, the string depth of any internal node in a suffix tree is equal to the node depth of the corresponding node in a trie. The results for the height of tries can be used as an upper bound for the height of compact trees in order to prove that the average space complexity of our index structure is $O(n \log n)$ and the average preprocessing time is $O(n \log^2 n)$. Moreover, we have these bounds *with high probability*. As mentioned above, we only deduce these results for the simple *Bernoulli* model for the sake of illustrating the techniques. Then we adopt the results for more general random sources as given in [Szp93] and use these results in our average case analysis.

3.6.1 Probabilistic Models of Strings

Throughout this section we consider a text to be a sequence of discrete random variables. We write $t = \{X_k\}_{k=1}^{\infty}$ for a one-sided infinite sequence of random variables and abbreviate it to X whenever the context clearly signals that we talk about such a sequence. We assume that the sequence $\{X_k\}$ is generated by a finite alphabet Σ, where $V = |\Sigma|$ is the fixed size of the alphabet. Finally, we always assume that there exists a probability measure for the random variables and we write $P(X_1^n) = Pr\{X_k = x_k, 1 \leq k \leq n, x_k \in \Sigma\}$.

Sequences are generated by some kind of probabilistic models, which we call information sources. According to the underlying models, the sources satisfy certain constraints. Here, to keep things simple, we assume that the sequence we analyze is generated by a memoryless source.

Definition 3.39 (Memoryless Source). *A sequence $\{X_k\}_{k=1}^{\infty}$ is generated by a memoryless source if the symbols $\{\sigma_1, \ldots, \sigma_V\}$ of the alphabet Σ occur independently of each other. Thus, the string $X_1, X_2, \ldots$ can be described as the outcome of an infinite sequence of Bernoulli trials in which $Pr\{X_j = \sigma_i\} = p_i$ and $\sum_{i=1}^{V} p_i = 1$.*

However, in many cases this model is not very realistic, and it is more convenient to use other information sources like the widely used *Markovian source*.

We consider a suffix trie for a text of length n to be built of the first n suffixes of a string X_1^{∞} generated by a memoryless source. Obviously, the suffixes $suf_1, \ldots, suf_n$ are dependent. The reader may have noticed that by Definition 2.6 such a trie has an infinite height. To overcome this, we make the following simplifications. We introduce (1)-tries. In a (1)-trie all suffixes are stored in leaf nodes. The access path from the root to a leaf contains a minimal prefix information. That means, when following a path from the root according to that information, we arrive at the corresponding external node. Thus, if for any node in a trie there is no more branching in its subtree, this node is obsolete in a (1)-trie. Intuitively a (1)-trie is a trie where all unary nodes between a leaf node and the parent of this leaf node in the corresponding compact suffix tree are eliminated.

This conception corresponds to the so-called *(b)-tries* [Szp93], where every external node can hold up to b suffixes – in our case $b = 1$. For the reader's convenience and to avoid mixing up notations, we keep apart tries and (1)-tries. For our purposes (1)-tries are very useful. As stated above, we are interested in investigating the height of suffix trees, and we do this by considering suffix (1)-tries. Clearly, the nodes that are eliminated in this new conception have no impact on the height of the corresponding compact suffix trees. On the other hand, this simplification is of great benefit in the analysis. With (1)-tries it is possible to define the height of a trie in terms of the *self-alignment* (see below) as well as to think of a trie being built of the first n suffixes of an infinite sequence, which is required by the probabilistic model.

3.6.2 Analysis of the Height of Suffix (1)-Tries

We define the height H_n of a (1)-trie. Again, we want to stress that we keep apart the heights of suffix trees or tries in the original Definitions 2.7 and 2.8 and the height of (1)-tries built of suffixes. In order to accomplish this, we write H_n exclusively for the height of (1)-tries and denote the height of suffix trees according to Definition 2.10 with h.

Definition 3.40 (Depth of a Suffix). *Let suf_m be a suffix and T be the corresponding suffix (1)-trie. The m-th depth $D_n(m)$ is the length of the path from the root of T to the external node that contains suf_m.*

Here, we refer to a path length as the number of nodes on that path excluding the starting node. Thus, the depth of a suffix is given by the node depth of the external node containing the suffix.

Definition 3.41 (Height of a Suffix (1)-Trie). *The height H_n of a suffix (1)-trie is the length of the longest depth, that is*

$$H_n = \max_{l \leq m \leq n} \{D_n(m)\} \tag{3.1}$$

The above definitions give the parameter of interest, namely the height of a suffix (1)-trie, a formal framework. Before analyzing it, we define the *self-alignment* of two suffixes and show how to use this concept in order to redefine the parameter.

Definition 3.42 (Self-Alignment). *For suffixes $suf_1, \ldots, suf_n$, the self-alignment C_{ij} between two suffixes suf_i and suf_j is the length of the longest common prefix of the two suffixes.*

In suffix (1)-tries the node depth (of internal nodes) is equal to the string depth, so it makes no difference whether we consider string depth or node depth. Having this in mind and using Definition 3.42, we get

Lemma 3.43 (Height of a (1)-Trie in Terms of the Self-Alignment). *The height of a (1)-trie in terms of the self-alignment is given as*

$$H_n = \max_{1 \leq i \leq j \leq n} \{C_{ij}\} + 1 \tag{3.2}$$

This instantly leads to

$$Pr\{H_n > k\} = Pr\{\max_{1 \leq i \leq j \leq n} \{C_{ij}\} \geq k\} \tag{3.3}$$

Remark 3.44. *Throughout the analysis infinite sequences X_1^∞ are considered. However, the application of our index for approximate pattern matching is clearly designed for finite strings. Nevertheless, the results we derive here and we adopt from Szpankowski can also be applied to finite strings. In our case we build the suffix (1)-trie from the first n suffixes of X. Obviously, X_{n+1}^∞ has no impact on H_n. To see why the results we adopt below can also be applied to finite strings, we refer to [Szp93]. Hence, in the following we can consider a suffix (1)-trie built from the first n suffixes of an infinite sequence to be equivalent – as far as H_n is concerned – to a suffix (1)-trie built of the n suffixes of a sequence of length n.*

We use Lemma 3.43 and Equation (3.3) to obtain results on the behavior of the height of suffix (1)-tries for strings that are created by a memoryless source. The results on this topic are mostly due to Devroye [Dev82], Pittel [Pit85], and Szpankowski [Szp90, Szp91, Szp93], and we mainly follow the exposition in [Szp00]. We begin with investigating the probability that two arbitrary suffixes suf_i and suf_j have a common prefix that is longer then k, that is we analyze $Pr\{C_{ij} \geq k\}$. Let $d = |i - j|$ and consider suf_i and suf_j. We distinguish the cases that $d \geq k$ and $d < k$. Clearly, if $d \geq k$, suf_i and $suf_j = suf_{i+d}$ are independent in the memoryless model as far as their longest common prefix is concerned. Consider the probability that two independent

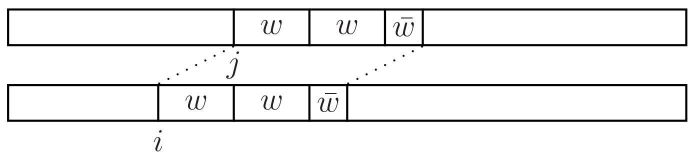


Figure 3.9: Lothaire's Lemma

strings have the same symbol x_i at a given position. Clearly, this probability is p_i^2, where p_i is the probability of character x_i being generated by the memoryless source. The probability that the two sequences share any character at that position is thus $P_2 = \sum_{i=1}^{V} p_i^2$. The probability that the two strings have exactly their first k characters in common has a geometric distribution, that is for all $i, j \in \{1, \ldots, n\}$ and $k \geq 0$

$$Pr\{C_{i,i+d} = k\} = P_2^k(1 - P_2) \tag{3.4}$$

Indeed, we can consider this as a Bernoulli experiment where the probability of success, that is the two strings have the same character at a given position, is given by P_2. Thus, the probability of failure is $(1 - P_2)$. The probability of $C_{i,i+d} \geq k$ is calculated as above, except that we do not require that we have a mismatch at position $k + 1$. Thus, in (1)-tries we find that

$$Pr\{C_{i,i+d} \geq k\} = P_2^k, \quad d \geq k \tag{3.5}$$

This is the simple case. However, the case $d < k$ is more demanding since in that case those characters involved in the longest common prefix of the two strings are dependent. Thus, we need to identify conditions under which k symbols starting at position i are the same as k symbols starting at position $i + d$, that is

$$X_i^{i+k-1} = X_{i+d}^{i+d+k-1} \tag{3.6}$$

The following simple combinatorial lemma provides a solution. The technique used here is called the *string-ruler approach*, which was introduced by Pittel [Pit85] and extended by Jaquet and Szpankowski [JS94]. The idea of this approach is to measure a correlation between words by another nonrandom word w.

Lemma 3.45 (Lothaire, 1982). *Let X_1^∞ be a string whose i-th and j-th suffixes suf_i and suf_j are X_i^∞ and X_j^∞, where $j - i = d > 0$. Let further Z be the longest common prefix of length $k \geq d$ of suf_i and suf_j. Then there exists a word $w \in \Sigma^d$ of length d so that*

$$Z = w^{\lfloor k/d \rfloor}\bar{w} \tag{3.7}$$

where $\bar{w}$ is a prefix of w and w^l is the string resulting from the concatenation of $l = \lfloor k/d \rfloor$ copies of word w

The formal proof of the lemma is omitted here, but the meaning of it is quite obvious and illustrated in Figure 3.6.2 . The existence of a prefix of length k of both suffixes is linked with the existence of the word wZ. Thus, the probability of having a common prefix of length k is equal to the probability of wZ, which is on its part the probability of any word of length d occurring in the form $w^{\lfloor k/d \rfloor+1}\bar{w}$. The lemma allows us to transform the probability for a self-alignment $C_{i,i+d}$ into a probability of some word wZ occurring in t. We obtain

$$Pr\{C_{ij} \geq k\} = \sum_{w \in \Sigma^d} P(w^{\lfloor k/d \rfloor+1}\bar{w}) = \tag{3.8}$$

$$\left(\sum_{i+1}^{V} p_i^{\lfloor k/d \rfloor+1}\right)^{d-r} \cdot \left(\sum_{i+1}^{V} p_i^{\lfloor k/d \rfloor+2}\right)^{r} \tag{3.9}$$

, where $r = k - d\lfloor k/d \rfloor$. (3.8) implies (3.9) because $d - r$ symbols of w must be repeated $l = \lfloor k/d \rfloor$ times, i.e. $l + 1$ occurrences in wZ while the last r symbols of w must occur $l + 2$ times. Observe that the probability of having the same symbol on a given position in l strings is given by $\left(\sum_{i=1}^{V} p_i^l\right) = P_l$.

Before concluding our analysis, we need an additional lemma.

Lemma 3.46. *Let* $p_i > 0$ *and* $\sum p_i \leq 1$ *for* $i \in \{1, \ldots, n\}$. *For all* $0 < s \leq t$, $s, t \in \mathbb{N}$, *the following holds.*

$$P_s^{\frac{1}{s}} = \left(\sum_{i=1}^{n} p_i^s\right)^{\frac{1}{s}} \geq \left(\sum_{i=1}^{n} p_i^t\right)^{\frac{1}{t}} = P_t^{\frac{1}{t}} \tag{3.10}$$

Proof. We show this by induction over n.

- Induction Start: $n = 2$
  We have to show that $(p_1^s + p_2^s)^{\frac{1}{s}} \geq (p_1^t + p_2^t)^{\frac{1}{t}}$.
  We set $p_1 = p$ and $p_2 = q$. Further, we define the function $f(x) = (p^x + q^x)^{\frac{1}{x}} = e^{\frac{1}{x}\ln(p^x+q^x)}$.
  Then $f'(x) = f(x) \cdot \left(\frac{1}{x} \cdot \frac{p^x \cdot \ln p + q^x \cdot \ln q}{p^x+q^x} - \frac{1}{x^2} \cdot \ln(p^x + q^x)\right)$.
  We assume that $p > q$ and that $x \geq 1$. We set $a = \frac{q}{p}$. Simplification of the above leads to $f'(x) = \left(\frac{1}{1+a^{-x}}\right)(\ln q - \ln p) + \left(1 + \frac{1}{x}\right) \cdot \ln p - \frac{1}{x} \cdot \ln(1 + a^x) \leq 0$.
  One can easily verify that the above is ≤ 0 as each of the three terms contributes a negative value to the sum. It follows that for $x \geq 1$, $f(x)$ is a decreasing function and thus the claim is proven for $n = 2$

- Induction Step: $n \to n + 1$: Suppose the claim holds for n.
  We have $\left(\sum_{i=1}^{n+1} p_i^s\right)^{\frac{1}{s}} = (p_1^s + \ldots + p_n^s + p_{n+1}^s)^{\frac{1}{s}}$.
  We set $p' = (p_n^s + p_{n+1}^s)^{\frac{1}{s}}$ and observe that $p' \leq 1$ and $p' \leq p_n + p_{n+1}$, then

$\left(\sum_{i=1}^{n+1} p_i^s\right)^{\frac{1}{s}} = (p_1^s + \ldots + p'^s)^{\frac{1}{s}}$.
Consequently, we have by induction hypothesis that
$\left(\sum_{i=1}^{n+1} p_i^s\right)^{\frac{1}{s}} \geq (p_1^r + \ldots + p'^r)^{\frac{1}{r}}$.
Again by induction hypothesis it follows that $p' = (p_n^s + p_{n+1}^s)^{\frac{1}{s}} \geq (p_n^r + p_{n+1}^r)^{\frac{1}{r}}$ and thus $(p_1^r + \ldots + p'^r)^{\frac{1}{r}} \geq (p_1^r + \ldots + p_n^r + p_{n+1}^r)^{\frac{1}{r}}$ and the claim is established.

□

We now estimate $Pr\{C_{i,i+d} \geq k\}$ for the case $d < k$. From Equation (3.9) we have that

$$Pr\{C_{i,i+d} \geq k\} = P_{l+1}^{d-r} \cdot P_{l+2}^{r} \tag{3.11}$$

, where $P_l = \sum_{i=1}^{V} p_i^l$ and

$$P_{l+1}^{d-r} \cdot P_{l+2}^{r} \leq P_2^{\frac{1}{2}[(d-r)(l+1)+r(l+2)]} = P_2^{\frac{k+d}{2}} \tag{3.12}$$

since $k = dl + r$. Equation (3.12) follows by Lemma 3.46.

Now, we can proof an upper bound for the height of (1)-tries. We use Boole's inequality (see Lemma A.6) and the above to obtain.

$$\begin{aligned} Pr\{\max_{i,d} C_{i,i+d} \geq k\} &\leq n \cdot \left(\sum_{d=1}^{k} Pr\{C_{i,i+d} \geq k\} + \sum_{d=k+1}^{n} Pr\{C_{i,i+d} \geq k\}\right) \\ &\leq n \cdot \left(\sum_{d=1}^{k} P_2^{\frac{k+d}{2}} + \sum_{d=k+1}^{n} P_2^{k}\right) \\ &\leq n \cdot \left(P_2^{\frac{k+1}{2}} \cdot (1 - \sqrt{P_2})^{-1} + n \cdot P_2^{k}\right) \end{aligned}$$

The second inequality comes from Lemma 3.46. We now set $k = 2(1+\epsilon)\log_{Q_2} n$. With Equation (3.13) we get

$$Pr\{H_n > 2(1+\epsilon)\log_{Q_2} n\} = Pr\{\max_{i,d}\{C_{i,i+d}\} \geq 2(1+\epsilon)\log_{Q_2} n\} \leq \frac{c}{n^{\epsilon}} \tag{3.13}$$

for some constant c and $Q_2 = P_2^{-1}$. This establishes the upper bound for convergence in probability. For definitions of stochastic convergences we refer to Section A.2 and to [Szp00, Shi84].

For our purposes it is essential that we can directly derive the height of a compact suffix tree in the average case along with a probability that this height is exceeded. We can use this in the average case analysis of the index structure.

Obviously, the height of a suffix (1)-trie is greater than or equal to the height of the corresponding compact suffix tree. Recall the correlations between string and node depth stated above. Thus, Equation (3.13) also holds for the probability of the height of a suffix tree exceeding the value k. We summarize our results in the following theorem.

Theorem 3.47 (Probabilistic Behavior of the Height of a Suffix Tree). *Let T be a suffix tree for the text t of length n generated by a memoryless source. Then for the height h of T the following holds.*

$$Pr\{h > 4\log_{Q_2} n\} \leq Pr\{\max_{i,d}\{C_{i,i+d}\} \geq 4\log_{Q_2} n\} \leq \frac{c}{n} \tag{3.14}$$

Proof. Directly from Equation (3.13) and $\epsilon = 1$. □

3.6.3 Generalization and Average Analysis of the Index Structure

We have explicitly deduced the probability of a suffix (1)-trie having height $\geq k$, where we have set k to be $O(\log n)$, for the case that the text t is created by a memoryless source. However, Szpankowski's results on the behavior of suffix trees apply to much more general cases, too. The principles in the probabilistic analysis of those cases are the same as in the case of sequences that are created by a memoryless source. Thus, without going into more detail at this point, we adopt Szpankowski's result and refer to the concerning papers [Szp93, Szp00].

Lemma 3.48 (Szpankowski). *Let X_1^n be a sequence that satisfies the strong mixing condition (see Definition A.2). Let h_2 be the Rènyis Order Entropy for $b = 1$ (compare Definition A.3). Note that h_2 is a constant. Then the following holds*

$$Pr\left\{H_n \geq (1+\epsilon)\cdot\frac{\log n}{h_2}\right\} \leq \frac{c\cdot(1+\epsilon)\log n}{n^\epsilon\cdot h_2} \tag{3.15}$$

Here, we have that H_n is the height of a suffix (1)-trie built over sequence X_1^n and h_2 is an explicit constant which depends on the type of probabilistic model. For the definition of the *strong mixing model*, we refer to the appendix (Definition A.2) and to the citations found there. Note, however, that the conditions demanded by this model are not very restrictive and hold for many interesting cases, including the Bernoulli model, see Section 3.6.2, and the widely used, popular *Markovian model*. We set $\epsilon = 1$ and obtain

$$Pr\left\{H_n \geq \frac{2}{h_2}\cdot\log n\right\} \leq \frac{2\cdot c}{h_2}\cdot\frac{\log n}{n} \tag{3.16}$$

We fix $c_h = \frac{2\cdot c}{h_2}$, which is obviously a constant. According to elementary probability theory it follows that

$$Pr\{H_n \leq \frac{2}{h_2}\cdot\log n\} \geq \left(1 - c_h\cdot\frac{\log n}{n}\right) \tag{3.17}$$

In order to use the obtained results, we reformulate them according to our needs. The size of error trees and the needed preprocessing time depends on the height of the corresponding suffix tree. In Definition 2.10 the height of a suffix tree was given by the greatest node depth of any external node. As explained in detail above, the bounds and probabilistic results obtained in this chapter for (1)-tries can be used to bound this parameter. We formalize this in the following lemma.

Lemma 3.49 (Probability Bound for the Height of a Suffix Tree). *Let h be the height of a suffix tree T. Then*

$$Pr\{h \geq \frac{2}{h_2} \cdot \log n\} \leq c_h \cdot \frac{\log n}{n} \tag{3.18}$$

Proof. The height h of a suffix tree is obviously $\leq H_n$, where H_n again denotes the height of the corresponding suffix (1)-trie. Thus, the probability for $H_n \geq \frac{2}{h_2} \cdot \log n$ is an upper bound for the probability of $h \geq \frac{2}{h_2} \cdot \log n$. The claim follows with Lemma 3.48 and the above discussion. □

Now, we are ready to analyze our index structure on average. We first consider the size of the aggregate error tree and the time needed to determine it. Recall Theorems 3.28 and 3.23. For some constants c_1 and c_2, the size of the index structure is given by $O(n \cdot h) = c_1 \cdot n \cdot h$ in dependence of h and $O(n^2) = c_2 \cdot n^2$ in the worst case. Thus, we have

$$\begin{aligned} \text{AverageSize(Index Structure)} = c_2 \cdot n^2 \cdot c_h \cdot \frac{\log n}{n} + c_1 \cdot n \cdot \log n \cdot \frac{2}{h_2} \cdot \left(1 - c_h \cdot \frac{\log n}{n}\right) = \\ = O(n \cdot \log n) \end{aligned} \tag{3.19}$$

For the preprocessing time of the index structure we have the bounds $O(n \cdot h^2)$ and $O(n^2)$, see again Theorems 3.28 and 3.23. So analogously we get

$$\text{AverageTime(Index Structure)} = O(n \log^2 n) \tag{3.20}$$

Thus, we can state the following theorems.

Theorem 3.50 (Average Size of the Index Structure). *The average size of the index structure for any sequence whose underlying probabilistic model satisfies the strong mixing condition is $O(n \log n)$.*

Proof. Equation (3.19) □

Theorem 3.51 (Average Preprocessing Time). *The average preprocessing time of the index structure for any sequence whose underlying probabilistic model satisfies the strong mixing condition is $O(n \log^2 n)$.*

Proof. Equation (3.20) □

The following also follows easily.

Theorem 3.52. *The size of the index structure is $O(n \log n)$ with high probability $1 - o(1)$.*

Theorem 3.53. *The preprocessing time for creating the index structure is $O(n \log^2 n)$ with high probability $1 - o(1)$.*

Chapter 4

Conclusion

We have introduced a new index structure, which facilitates linear determination of all substrings of a text that match an arbitrary input pattern with one mismatch. We have shown that the size of this index structure is $O(n \log n)$ and that the preprocessing time needed is $O(n \log^2 n)$, both on average and with high probability.

Another nice property of our index structure is that it is conceptually very easy. Most of the proofs in this work are obvious, and the whole method is very intuitive. The core elements of our index structure are commonly known data structures and algorithms, which are extended by our approach.

The following results from previous work are incorporated in our method. We use suffix trees and their linear construction as starting point. Our algorithm for building the index structure uses the constant time lca-query solution. The time and space complexity of the index structure depends on the height of the corresponding suffix tree, which is proven to be $O(n \log n)$ with high probability for a variety of probabilistic models.

Consider the difference between our method and building a full index on all strings that match any substring of the text with one mismatch. Such an index is e.g. a Σ^+-tree that is built by all suf_i's, where suf_i' is the i-th suffix of t and one character is substituted, i.e. $suf_i' = \alpha a \beta$ and $suf_i = \alpha b \beta$ with $a \neq b$. The size of such a tree is obviously $O(n^2)$, but it is possible to locate any occurrence of a given pattern in linear time with respect to the pattern size.

The key observation that makes our approach superior to that is the following. When considering the suffix tree for the text, mismatches that occur on edges between internal and external nodes do not have to be considered. Since there is no further branching on such edges, mismatches can be verified with the knowledge of the suffix tree alone. If we recall that the height of a suffix tree is $O(\log n)$ with high probability, this means that we are basically only covering strings of length $O(\log n)$ that match some substring of the text with one error, i.e. substrings of the text that are in that part of the suffix tree that is relevant. Additionally, only errors after nodes are considered, which reduces the space even more.

There are several open problems that need future attention. First of all it seems to be interesting to expand the approach to a greater, constant number of errors. We conjecture that for the case of two allowed mismatches, an extension of our approach provides an index of size $O(n \log^2 n)$, still allowing for linear searching. More generally, iterating this could yield an index with linear lookup times for a constant number of mismatches under the penalty of additional $\log$-factors. Furthermore, investigating this poses some interesting questions concerned with probabilistic analysis of strings and promises further theoretical insights. However, even for the one mismatch case, we still have to develop a preprocessing method that is linear in the size of the index structure, i.e. $O(nh)$.

We have not performed empirical experiments with our method to investigate its performance in practice. In addition to that we are looking for a way of adapting the index to handle also insertions and deletions. The worst case size of the index structure is still an open problem. Given a suffix tree whose height is linear in the text size, the index structure has size $O(n^2)$. However, we hope that by finding an intelligent way of storing the aggregate error tree this bound can be improved. Texts that have a suffix tree of linear height have certain repetitious elements, which leads to a lot of redundant information in the AET, i.e. isomorphic or “nearly” isomorphic substructures of the tree. Hopefully, this or other facts can be exploited.

Another basic approach is to adapt the index structure to suffix arrays instead of trees. This could further improve the space requirements and maybe lead to practically efficient applications.

Bibliography

[AG97] Alberto Apostolico and Zvi Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[AKL+99] Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Indexing and dictionary matching with one error. In *Workshop on Algorithms and Data Structures*, pages 181–192, 1999.

[BFC00] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, may 2000.

[BGW00] Adam L. Buchsbaum, Michael T. Goodrich, and Jeffery Westbrook. Range searching over tree cross products. In *European Symposium on Algorithms*, pages 120–131, 2000.

[Bil65] P. Billingsley, editor. *Ergodic Theory and Information*. John Wiley & Sons, New York, 1965.

[BYN99] Ricardo A. Baeza-Yates and Gonzalo Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

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

[Coo94] Necia Grant Cooper. *The Human Genome Project: Deciphering the Blueprint of Heredity*. University Science Books, Mill Valey, California, 1994.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 1964.

[Dev82] Luc Devroye. A note on the average depth of tries. *Computing, Springer*, 28(4):367–371, 1982.

[Far97] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS97)*, 1997.

[GMRS02] A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Indexing structure for approximate string matching. Technical Report 169, University of Palermo, 2002.

[GMRS03] A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Indexing structures for approximate string matching. In *Lecture Notes in Computer Science*, volume 2653, pages 140–151, 2003.

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

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, pages 147–160, 1950.

[Hec78] P. Heckel. A technique for isolating differences between files. *Communicaations of the ACM*, 21:264–268, 1978.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, may 1984.

[JS94] Philippe Jacquet and Wojciech Szpankowski. Autocorrelation on words and its applications: Analysis of suffix trees by string-ruler approach. *JCTA: Journal of Combinatorial Theory, Series A*, 66, 1994.

[KSP99] Dong Kyue Kim, Jeong Seop Sim, and Kunsoo Park. Suffix trees for integer alphabets revisited. In *3rd Korea-Japan Joint Workshop on Algorithms and Computation WAAC'99*, pages 171–178, Seoul, Corea, 1999.

[Les94] A. M. Lesk. Computational molecular biology. *Enyclopedia of Computer Science and Technology*, 31:101–165, 1994.

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

[LS97] T. Luczak and W. Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Trans. on Inf. Theory*, 43:1439–1451, 1997.

[LT97] Daniel Lopresti and Andrew Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

[Maa00] Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1848 of *LNCS*, pages 320–334. Springer, jun 2000.

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

[MP68] Marvin L. Minsky and S. Papert. *Perceptronics*. MIT Press, Cambridge, Massachusetts, 1968.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666, 2002.

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

[Nav98] G. Navarro. *Approximate Text Searching*. PhD thesis, Univ. of Chile, 1998.

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

[NBY99] Gonzalo Navarro and Ricardo Baeza-Yates. A new indexing method for approximate string matching. In *Proc. of the 10th Sym. on Combinatorial Pattern Matching (CPM)*, volume 1645 of *LNCS*, pages 163–185. Springer, 1999.

[NBY00] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. Discret. Algorithms*, 1(1):205–239, 2000.

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

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[Ove88] Mark H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9(2):254–275, 1988.

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

[Shi84] A.N. Shirjayev. *Probability*. Springer, 1984.

[SK83] David Sankoff and Joseph B. Kruskal. *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*. Addison-Wesley, 1983.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, dec 1988.

[Szp90] Wojciech Szpankowski. Patricia tries again revisited. *Journal of the ACM*, 37(4):691–711, oct 1990.

[Szp91] Wojciech Szpankowski. A characterization of digital search trees from the successful search viewpoint. *Theoretical Computer Science*, 85:117–134, 1991.

[Szp93] Wojciech Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. COMPUT.*, 22(6):1176–1198, dec 1993.

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

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

[Wat95] Micheal S. Waterman. *Introduction to Computational Biology: Maps, sequences and genomes*. Chapman and Hall, 1995.

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

Appendix A

Appendix

A.1 Probabilistic Models of Strings

Definition A.1 (Mixing Model). *The sequence $\{X_k\}_{k=-\infty}^{\infty}$ satisfied the so-called mixing condition [Bil65] if there exist two constants $0 < c_1 < c_2$ and an integer d such that for all $-\infty \leq m \leq m+d \leq n$ the following holds*

$$c_1 Pr\{\mathcal{B}\}Pr\{\mathcal{B}\} \leq Pr\{\mathcal{AB}\} \leq c_2 Pr\{\mathcal{B}\}Pr\{\mathcal{B}\} \tag{A.1}$$

where $\mathcal{A} \in \mathcal{F}_{-\infty}^{m}$ and $\mathcal{B} \in \mathcal{F}_{m+d}^{\infty}$ and $\mathcal{F}_m^n$ is the σ-field generated by $\{X_k\}_{k=m}^{n}$ for $m \leq n$

Definition A.2 (Strong Mixing Model). *The sequence $\{X_k\}_{k=-\infty}^{\infty}$ satisfies the so-called strong α-mixing condition if equation (A.1) is replaced by*

$$(1-\alpha(d))Pr\{\mathcal{B}\}Pr\{\mathcal{B}\} \leq Pr\{\mathcal{AB}\} \leq (1+\alpha(d))Pr\{\mathcal{B}\}Pr\{\mathcal{B}\} \tag{A.2}$$

where $\alpha(d)$ is a function such that $\alpha(d) \to 0$ as $d \to \infty$

Both models characterize the dependency between parts of the sequence that are d positions apart. The first Model A.1 means, that there is only a weak dependency between the considered parts of the sequence, whereas Model A.2 states that the dependency is getting weaker as d grows. Note that e.g. the Markovian model satisfies A.2.

Definition A.3 (Rényis Order Entropy). *For $-\infty < b < \infty$ define the b-th order Rényi entropy as*

$$h_2^{(b)} = \lim_{n\to\infty} \frac{-\log(E\{P^b(X_1^n)\})^{-1}}{(b+1)n} = \lim_{n\to\infty} \frac{-\log(\sum_{X_1^n} P^{b+1}(X_1^n))^{1/(b+1)}}{n} \tag{A.3}$$

A.2 Types of Stochastic Convergence

For details on this subject we refer to [Shi84].

Definition A.4 (Convergence in Probability). *The sequence $X_1, X_2, \ldots$ converges in probability to the random variable X if for every $\epsilon > 0$*

$$P\{|X_n - X|\} \to 0, n \to \infty \tag{A.4}$$

Definition A.5 (Almost Sure Convergence). *The sequence $X_1, X_2, \ldots$ converges almost surely (with probability one, almost everywhere) to the random variable X if*

$$P\{\omega : X_n \nrightarrow X\} = 0 \tag{A.5}$$

i.e. if the set of sample points ω for which $X_n(\omega)$ does not converge to X has probability zero.

A.3 Inequalities

Lemma A.6 (Boole's Inequality). *Let $A_1, \ldots A_n$ be arbitrary events. Then*

$$Pr\{\bigcup_{i=1}^{n} A_i\} \leq \sum_{i=1}^{n} Pr\{A_i\} \tag{A.6}$$