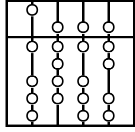
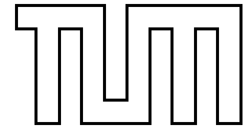


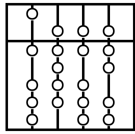
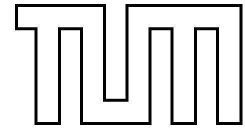
FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN



Lehrstuhl für Effiziente Algorithmen

Analysis of Algorithms and Data Structures for Text Indexing

Moritz G. Maaß


FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN


Lehrstuhl für Effiziente Algorithmen

Analysis of Algorithms and Data Structures for Text Indexing

Moritz G. Maaß

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Dr. h.c. mult. Wilfried Brauer
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Ernst W. Mayr
	2.	Prof. Robert Sedgewick, Ph.D. (Princeton University, New Jersey, USA)

Die Dissertation wurde am 12. April 2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26. Juni 2006 angenommen.

Abstract

Large amounts of textual data like document collections, DNA sequence data, or the Internet call for fast look-up methods that avoid searching the whole corpus. This is often accomplished using tree-based data structures for text indexing such as tries, PATRICIA trees, or suffix trees. We present and analyze improved algorithms and index data structures for exact and error-tolerant search.

Affix trees are a data structure for exact indexing. They are a generalization of suffix trees, allowing a bidirectional search by extending a pattern to the left and to the right during retrieval. We present an algorithm that constructs affix trees on-line in both directions, i.e., by augmenting the underlying string in both directions. An amortized analysis yields that the algorithm has a linear-time worst-case complexity.

A space efficient method for error-tolerant searching in a dictionary for a pattern allowing some mismatches can be implemented with a trie or a PATRICIA tree. For a given mismatch probability q and a given maximum of allowed mismatches d, we study the average-case complexity of the number of comparisons for searching in a trie with n strings over an alphabet of size σ. Using methods from complex analysis, we derive a sublinear behavior for $d < \log_\sigma n$. For constant d, we can distinguish three cases depending upon q. For example, the search complexity for the Hamming distance is $\sigma(\sigma-1)^d/((d+1)!) \log_\sigma^{d+1} n + O(\log^d n)$.

To enable an even more efficient search, we utilize an index of a limited d-neighborhood of the text corpus. We show how the index can be used for various search problems requiring error-tolerant look-up. An average-case analysis proves that the index size is $O(n \log^d n)$ while the look-up time is optimal in the worst-case with respect to the pattern size and the number of reported occurrences. It is possible to modify the data structure so that its size is bounded in the worst-case while the bound on the look-up time becomes average-case.

Acknowledgments

First, I thank my advisor Ernst W. Mayr for his helpful guidance and generous support throughout the time of researching and writing this thesis. Furthermore, I am also thankful to the current and former members of the *Lehrstuhl für Effiziente Algorithmen* for interesting discussions and encouragement, especially to Thomas Erlebach, Sven Kosub, Hanjo Täubig, and Johannes Nowak. Lastly, I am grateful to Anja Heilmann and my mother for proofreading the final work.

Contents

1 Introduction **1**
1.1 From Suffix Trees to Affix Trees 2
1.2 Approximate Text Indexing 4
1.2.1 Tree-Based Accelerators for Approximate Text Indexing . . . 5
1.2.2 Registers for Approximate Text Indexing 7
1.3 Thesis Organization 8
1.4 Publications 9

2 Preliminaries **11**
2.1 Elementary Concepts 11
2.1.1 Strings 11
2.1.2 Trees over Strings 13
2.1.3 String Distances 14
2.2 Basic Principles of Algorithm Analysis 15
2.2.1 Complexity Measures 15
2.2.2 Amortized Analysis 16
2.2.3 Average-Case Analysis 17
2.3 Basic Data Structures 19
2.3.1 Tree-Based Data Structures for Text Indexing 19
2.3.2 Range Queries 22
2.4 Text Indexing Problems 25
2.5 Rice's Integrals 26

3 Linear Construction of Affix Trees **31**
3.1 Definitions and Data Structures for Affix Trees 31
3.1.1 Basic Properties of Suffix Links 31
3.1.2 Affix Trees 34
3.1.3 Additional Data for the Construction of Affix Trees 35
3.1.4 Implementation Issues 39
3.2 Construction of Compact Suffix Trees 40
3.2.1 On-Line Construction of Suffix Trees 40
3.2.2 Anti-On-Line Suffix Tree Construction with Additional Information 43
3.3 Constructing Compact Affix Trees On-Line 47
3.3.1 Overview 47
3.3.2 Detailed Description 49

3.3.3 An Example Iteration . . . 51
3.3.4 Complexity . . . 54
3.4 Bidirectional Construction of Compact Affix Trees . . . 54
3.4.1 Additional Steps . . . 55
3.4.2 Improving the Running Time . . . 56
3.4.3 Analysis of the Bidirectional Construction . . . 57

4 Approximate Trie Search 63
4.1 Problem Statement . . . 64
4.2 Average-Case Analysis of the LS Algorithm . . . 65
4.3 Average-Case Analysis of the TS Algorithm . . . 67
4.3.1 An Exact Formula . . . 67
4.3.2 Approximation of Integrals with the Beta Function . . . 69
4.3.3 The Average Compactification Number . . . 77
4.3.4 Allowing a Constant Number of Errors . . . 79
4.3.5 Allowing a Logarithmic Number of Errors . . . 84
4.3.6 Remarks on the Complexity of the TS Algorithm . . . 87
4.4 Applications . . . 91

5 Text Indexing with Errors 93
5.1 Definitions and Data Structures . . . 95
5.1.1 A Closer Look at the Edit Distance . . . 95
5.1.2 Weak Tries . . . 99
5.2 Main Indexing Data Structure . . . 101
5.2.1 Intuition . . . 102
5.2.2 Definition of the Basic Data Structure . . . 102
5.2.3 Construction and Size . . . 104
5.2.4 Main Properties . . . 106
5.2.5 Search Algorithms . . . 107
5.3 Worst-Case Optimal Search-Time . . . 110
5.4 Bounded Preprocessing Time and Space . . . 113

6 Conclusion 115

Bibliography 118

Index 129

Figures and Algorithms

2.1 Examples of Σ^+-trees 21
2.2 Illustration of the linear-time Cartesian tree algorithm 23

3.1 Suffix trees, suffix link tree, and affix trees for `ababc` 32
3.2 The affix trees for `aabababa` and for `aababababaa`. 40
3.3 The procedure canonize() 42
3.4 Constructing ST(`acabaabac`) from ST(`acabaaba`) 44
3.5 The procedure decanonize() 45
3.6 The procedure update-new-suffix() 46
3.7 Constructing ST(`cabaabaca`) from ST(`abaabaca`) 48
3.8 The function getTargetNodeVirtualEdge() 49
3.9 Constructing AT(`acabaabac`) from AT(`acabaaba`), suffix view . . . 52
3.10 Constructing AT(`acabaabac`) from AT(`acabaaba`), prefix view . . . 53

4.1 The LS algorithm 64
4.2 The TS algorithm 65
4.3 Illustration for the proof of Lemma 4.6 76
4.4 Location of the poles of g(z) in the complex plane 81
4.5 Illustration for Theorem 4.14. 88
4.6 Parameters for selected comparison-based string distances 91

5.1 The relevant edit graph for `international` and `interaction` . . . 96
5.2 Examples of a compact and weak tries 101

Chapter 1

Introduction

Computers, although initially invented for numeric calculations, have changed the way that text is written, processed, and archived. Even though many more texts than initially expected or hoped for are still printed to paper, writing is mostly done electronically today. Having text available in a digital form has many advantages: It can be archived in very little space, it can be reproduced with little effort and without loss in quality, and it can be searched by a computer. The latter is a great improvement over previous methods especially because it is not necessary to create a fixed set of terms that are used to index the documents. When employing a computer, one commonly allows every word to be used for searching, often called full-text search. Efficient methods for searching in texts are studied in the realm or pattern matching. Pattern matching is already a rather mature field of research with numerous textbooks available [CR94, AG97, Gus97]; the basic algorithms are also part of standard books on algorithms [CLR90, GBY91]. Searching a text for an arbitrary pattern is usually almost as fast as reading the text (especially, if reading is hampered in speed because the text is stored on a slow media such as a hard disk).

The ease with which electronic documents can be stored has also lead to an enormous increase in the amount of textual data available. The Internet (in particular the World Wide Web), for example, is a tremendous and growing collection of text documents. The widely used search engine Google reports to index more than eight billion web pages.[1] Another type of textual data is biological sequence data (e.g., DNA sequences, protein sequences). A popular database for DNA sequences, GenBank, was reported to store over 33 billion bases for over 140 000 species in August 2003 and to grow at a rate of over 1700 new species per month [BKML$^+$04].

The sheer size of these text collections makes on-line searches unfeasible. Therefore, a lot of research has been devoted to methods for efficient text retrieval [FBY92] and text indexes. A text index is a data structure prepared for a document or a collection of documents that facilitates efficient queries for a search pattern. These queries can have different types. For a query with a single pattern, one can search, e.g., for all occurrences of a search string, for all occurrences of an element of a set of strings, or for all occurrences of a substring that has length twenty and contains at least ten equal characters—the possibilities for different criteria are endless and usually the result of

[1] These numbers were taken from `http://www.google.com/corporate/facts.html` as of March 2005.

specific applications. For the classical problem of searching for all occurrences of a given string (the "pattern") in a preprocessed text, many algorithms and data structures have been proposed. The most prominent data structure is the suffix tree, which can be constructed in linear time [Wei73, McC76, Ukk95, Far97]. It allows finding all occurrences of a pattern in time proportional to the pattern's length and the number of outputs. Due to limitations of space, the suffix array was introduced [MM93] as a space efficient replacement for the suffix tree with slightly worse construction and look-up complexities. Recent developments, however, yield a linear construction time [KSPP03, KA03, KS03] and achieve an optimal query time [AKO02, AOK02, AKO04]. In practice, construction methods with non-linear asymptotic running times seem to be even faster [LS99, BK03, MF04]. For tight space requirements, compressed variants of the suffix array have been suggested [GV00, FM00]. The practical relevance of the suffix tree has thus been reduced. Nevertheless, because of its clear structure, it often serves as a paradigm for the design of new algorithms (e.g., in [NBY00]).

While the computer allows creating and archiving large collections of text efficiently, it sometimes also leads to a less careful handling and revising of documents. Errors may also arise from mistakes or measuring problems during the experimental generation. Moreover, in a biological context, error-tolerant matching is useful even for immaculate data, e.g., for similarity searches. The field of approximate pattern matching deals with this problem of searching strings allowing errors. The most commonly used models are Hamming distance [Ham50] and edit distance, also known as Levenshtein distance [Lev65]. For so called on-line searches, where no preprocessing of the document corpus is done, a variety of algorithms is available for many different error models. A survey containing more than 120 references is given in [Nav01]. The research on text indexes allowing errors is by far not as mature.

We review the suffix tree as a paradigm for exact text indexing and as the basis for affix trees in the next section. We then turn to approximate text indexing, a very active field of research with still quite a few open questions.

1.1 From Suffix Trees to Affix Trees

The suffix tree, also sometimes referred to as subword tree, is a rooted tree allowing access to all suffixes (and thus to all substrings) of a text from the root of the tree. When annotating the suffix numbers at the leaves, the subtree below a path corresponding to a substring of the text is a compact representation of all occurrences of this substring. This principle allows an exact search in optimal time. The first linear time construction algorithm for suffix trees was given by Weiner [Wei73]. McCreight described a simpler and more efficient algorithm [McC76]. A renewed interest in pattern matching has also lead to the conceptually most elegant construction algorithm by Ukkonen [Ukk95]. It was later shown by Giegerich and Kurtz that Ukkonen's and McCreight's algorithms are almost the same on an abstract level of operations. All three algorithms consider strings over an alphabet of constant size. An algorithm to construct suffix trees over integer alphabets was proposed by Farach [Far97]. This algorithm uses a very different approach, dividing the underlying string into odd and even positions. (Note that this approach is also taken by the linear time suffix array construction algorithms.) The

three previous algorithms rely on auxiliary edges called suffix links instead.

Suffix links introduce additional structure themselves. The analysis of this structure was the starting point for the development of affix trees. Before we go into more detail, let us give a high level description of suffix trees; the formal definitions are introduced in Chapter 2. A suffix tree contains all suffixes of a string t in such a way that from the root of the tree we can start reading any substring of t. This is done by first choosing a branch upon the first character, then upon the second and so on. Thus, all substrings starting with the character `a` are stored in a branch labeled `a`. Nodes that do not present a choice, i.e., which have only one outgoing edge, are usually compressed. That is, the nodes are eliminated by joining the incoming and outgoing edge into one. Such suffix trees are called compact. Each node in a suffix tree represents a string and is the root of the subtree containing all substrings of t starting with this string. Passing along an edge to another node lengthens the represented string. Suffix links are used to move from one node to another so that the represented string is shortened at the front. This is extremely useful because two suffixes of a string always differ by the characters at the beginning of the longer suffix. Thus, successive traversal of suffixes can be sped up greatly by the use of suffix links.

Giegerich and Kurtz [GK97] have shown that there is a strong relationship between the suffix tree built from the reverse string (often called reverse prefix tree) and the suffix links of the suffix tree built from the original string. The suffix links of a suffix tree form a tree by themselves. In essence, this tree partially resembles the suffix tree of the reverse string. Atomic suffix trees contain a node for every substring of t. Here, the suffix link structure is even identical (see Section 3.1.1 for details).

A similar relationship was already shown by Blumer et al. [BBH$^+$87] for compact directed acyclic word graphs (c-DAWG). A c-DAWG is essentially a suffix tree where isomorphic subtrees are merged. As shown there, the directed acyclic word graphs (DAWG) contains all nodes of the compact suffix tree and uses the edges of the reverse prefix tree as suffix links. Because the nodes of the c-DAWG are invariant under reversal of the string, one can gain a symmetric version of the c-DAWG by adding the suffix links as reverse edges. Although not stated explicitly, Blumer et al. [BBH$^+$87] thus already observed the dual structure of suffix trees as stated above.

Because the suffix links of the compact suffix tree already form part of the suffix tree for the reverse string, it is natural to ask whether the suffix tree can be augmented to represent both structures. This would also accentuate even more structure of the underlying string. Such a structure was introduced and christened affix trees by Stoye [Sto95] (see also [Sto00]). He also gave a first algorithm for constructing compact affix trees on-line, but the running time of the algorithm presented is not linear. In Chapter 3, we describe a linear-time algorithm for the construction of affix trees. The structure of our algorithm is very close to that of Stoye's algorithm, but we introduce some vital new elements. Moreover, we put a stronger emphasis on the derivation from Ukkonen's and Weiner's suffix tree construction algorithms.

The core problem for achieving linear complexity and the reason why Stoye's algorithm did not have a linear running time is that the classical algorithms by Ukkonen or McCreight expect suffix links to be of length one. That is, suffix links are expected to point from a node representing a certain string to a node representing exactly the string that results from removing the first character. As can easily be seen in Figure 3.1, this is the case for suffix trees but not for affix trees (nodes `aba` and `abab` in Figure 3.1(d)

and nodes ba and cbab in Figure 3.1(e)). It seems that the problem cannot be solved by traversing the tree to find the appropriate atomic suffix links. Stoye himself gave an example where this approach might even lead to quadratic behavior. We will use additional data collected in paths to establish a view of the affix tree as if it were the corresponding suffix tree (the paths correspond to the boxed nodes in Figure 3.1). This will lead to a linear-time construction algorithm.

1.2 Approximate Text Indexing

In approximate text indexing, we index a single text or multiple texts to find approximate matches of a query string, i.e., matches allowing some errors. The text or the collection of texts that the index is built for is called the document corpus. We only deal with the case of a single query string called the pattern. There are two major indexing tasks: text indexing and dictionary indexing.[2] For the first problem, a text is indexed to query for occurrences of substrings of the text. For the second, a dictionary of strings is indexed as to find all matching entries for a query. We discuss more variants in Section 2.4 (e.g., the case of multiple texts), but they can usually be solved with the same methods once the two basic problems are solved. In the following, let the size of document corpus be n and the size of the pattern be m. The basic idea of text indexing is to spend some effort in preparing an additional data structure, the index, from a text in order to accelerate subsequent searches for a pattern.

For exact text indexing, an optimal query time linear in the size of the pattern and the number of outputs can be reached. From a practical point of view, a query time involving an additional term logarithmic or polylogarithmic in the size of the text may still be reasonable. Similar complexities are frequently found in other search data structures. For example, a search in a red-black tree (see, e.g., [CLR90]) for n elements takes time of $O(\log n)$.

Besides the query time, the size of an index is another very important parameter. We already mentioned that the space demands of the suffix tree—although linear in the size of the text—are still too large for some applications, which has lead to the introduction of alternatives like suffix arrays. Even for these, compressed variants are studied. This and the ever-growing collections of text documents underline the importance of the parameter size.

The third parameter, aside from look-up time and index size, is the preprocessing time (and possibly space). This parameter is usually the least important because text indexes are built once on a relatively static text corpus to facilitate many queries. For example, the first suffix array algorithm by Manber and Myers [MM93] had a preprocessing time of $O(n \log n)$ for texts of n characters. Although this is worse than the $O(n)$ construction time for suffix trees, suffix arrays gained popularity quickly because they consumed much less space. In that case, even the worse look-up time did not hinder the success.

There appear various definitions of approximate text indexing in the literature. Besides the existence of many different error models (see Section 2.1.3), the definition

[2] The dictionary *indexing* problem is not to be confused with the dictionary *matching* problem. In the latter, a dictionary of patterns is indexed and queried by a text. For each position in the text all matching patterns from the dictionary are sought. The exact version can be solved with automatons [AC75]. For the approximate version see [FMdB99, AKL$^+$00, BGW00, CGL04].

of "index" is not clear. The broader definition just requires the index to "speed up searches" [NBYST01], whereas a stricter approach requires to answer on-line queries "in time proportional to the pattern length and the number of occurrences" [AKL$^+$00]. In our opinion, a slightly weakened definition that requires an index to give efficient worst-case guarantees on the search time is most appropriate. For the sake of making a clear distinction, we call the methods giving such guarantees *registers* and the other methods *accelerators*.[3] Classifying our work from this perspective, we present the average-case analysis of the effect of using a trie as an accelerator for dictionary indexing in Chapter 4. In Chapter 5, we first describe a register with a bounded average-case size for arbitrary patterns and then a combination of a register for short patterns with an accelerator for long patterns.

A survey on accelerators is given in [NBYST01]. The available methods are classified by the search approach and the data structures. The search approaches are classified into neighborhood generation and partitioning. The basic data structures are suffix trees, suffix arrays, complete q-gram tables, and sampled q-gram tables. As already mentioned, suffix arrays are very similar to suffix trees. We handle these and other tree-based structures in the next section. In Section 1.2.2, we turn our attention to registers. Further references concerning q-gram-based approaches can be found in [Kär02].

A very different approach is taken by Chávez and Navarro [CN02]. Using a metric index they achieve a look-up time $O(m \log^2 n + m^2 + \mathrm{occ})$ with an index of size $O(n \log n)$ with $O(n \log^2 n)$ construction time, where all complexities are achieved on average.

Another somewhat different approach is taken by Gabriele et al. [GMRS03]. For a restricted version of Hamming distance allowing d mismatches in every window of r characters, they describe an index that has average size $O(n \log^l n)$, for some constant l and average look-up time $O(m + \mathrm{occ})$. The idea is based on dividing the text into smaller pieces so that strings of length m have unique occurrences. At each level, all strings within a specified distance are generated and indexed to identify possible locations of the pattern. These are then searched using standard methods. Therefore, the approach classifies as an accelerator or a filter.

A special case of dictionary indexing is the nearest neighbor problem, which asks for the closest string to a query among a collection of n strings of the same length m. The specific problem has its origin in computational geometry. A very early definition can be found in [MP69]. A survey on this problem is given in [Ind04]. The work also discusses some lower bounds (see also [BOR99, BR00, BV02] and [CCGL99]), which—at least for [BV02]—transfer to the approximate dictionary indexing problem showing that, under certain assumptions, any algorithm using an index of size $2^{(nm)^{o(1)}}$ and $(nm)^{o(1)}$ additional working space needs at least $\Omega(m \log m)$ time for a query.

1.2.1 Tree-Based Accelerators for Approximate Text Indexing

Digital trees have a wide range of applications and belong to the most studied data structures in computer science. They have been around for years; for their usefulness

[3] None of these terms appear in the literature. We introduce them here in order to better classify different approaches in this section only. The term "register" seemed appropriate because worst-case methods usually work by reading off the occurrences of a pattern after computing certain start points. On the other hand, average-case methods often just reduce the number of locations where to check for an occurrence, thus accelerating an on-line search algorithm.

and beauty of analysis, they have received a lot of attention. Using tree (especially trie or suffix tree) traversals for indexing problems is a common technique. For instance, in computer linguistics one often needs to correct misspelled input. Schulz and Mihov [SM02] pursue the idea of correcting misspelled words by finding correctly spelled candidates from a dictionary implemented as a trie or an automaton. They build an automaton for the input word and traverse the trie with it in a depth first search. The search automaton is linear in the size of the pattern if only a constant number of errors is allowed. A similar approach has been investigated by Oflazer [Ofl96], except that he directly calculates edit distance instead of using an automaton.

Flajolet and Puech [FP86] analyze the average-case behavior of partial matching in k-d-tries. A pattern in k domains with s specified values and $(k-s)$ don't care symbols is searched. Each entry in a k-dimensional data set is represented by the binary string constructed by concatenating the first bits of the k domain values, the second bits, the third bits, and so forth. Using the Mellin transform, Flajolet and Puech prove that the average search time in a database of n entries under the assumption of an independent uniform distribution of the bits is $O(n^{1-s/k})$. The analysis is extended by Kirschenhofer et al. [KPS93] to the asymmetric case, where the exponent depends upon k, s, and the bit probability p (an explicit formula cannot be given). In the same paper, the authors also derive asymptotic results for the variance, thus proving convergence in probability. In terms of ordinary strings, searching in a k-d-trie corresponds to matching with a fixed mask of width k containing s don't cares that is iterated through the pattern. It is possible to randomize this fixed mask which yields "relaxed k-d-trees", which are analyzed with similar results by Martinéz et al. [MPP01].

Baeza-Yates and Gonnet [BYG96] study the problem of searching regular expressions in a trie. The regular expression is used to build a deterministic finite state automaton, whose size depends only upon the size of the query (although it is possibly exponential in the query size). The automaton is simulated on the trie, and a hit is reported every time a final state is reached. Extending the average-case analysis of Flajolet and Puech [FP86], the authors show that the average search time depends upon the largest eigenvalue (and its multiplicity) of the incidence matrix of the automaton. As a result, they prove that only a sublinear number of nodes of the trie is visited.

In another article, Baeza-Yates and Gonnet [BYG90, BYG99] study the average cost of calculating an all-against-all sequence matching. Here, any substrings that match each other with a certain (fixed) number of errors are sought. With the use of tries, the average time is shown to be subquadratic.

Apostolico and Szpankowski [AS92] note that suffix trees and tries for independent strings asymptotically do not differ too much under the symmetric Bernoulli model. This conjecture is hardened by Jacquet and Szpankowski [JS94] and by Jacquet et al. [JMS04]. Therefore, it seems reasonable to expect similar results on tries and suffix trees (as is done by Baeza-Yates and Gonnet [BYG96]).

For approximate indexing (with edit distance), Navarro and Baeza-Yates [NBY00] describe a method that flexibly partitions the pattern in pieces that can be searched in sublinear time in the suffix tree for a text.[4] For an error rate $\alpha = \frac{d}{m}$, where m

[4] For an actual implementation, Navarro and Baeza-Yates suggest to use suffix arrays instead of suffix trees. This is an example of how the suffix tree, although possibly inferior to the suffix array, plays an important role as a paradigm for the design and analysis of algorithms.

is the pattern length and d the allowed number of errors, they show that a sublinear-time search is possible if $\alpha < 1 - e/\sqrt{|\Sigma|}$, thereby partitioning the pattern into $j = (m+d)/(\log_{|\Sigma|} n)$ pieces. The threshold $1 - e/\sqrt{|\Sigma|}$ plays two roles, it gives a bound on the search depth in a suffix tree and it gives a bound on the number of verifications needed. In Navarro [Nav98], the bound is investigated more closely. It is conjectured that the real threshold, where the number of matches of a pattern in a text decreases exponentially fast in the pattern length, is $\alpha = 1 - c/\sqrt{|\Sigma|}$ with $c \approx 1.09$. Higher error rates make a filtration algorithm useless because too many positions have to be verified.

More careful tree traversal techniques can lower the number of nodes that need to be visited. This idea is pursued by Jokinen and Ukkonen [JU91] (on a DAWG), although no better bound than $O(nm)$ is given for the worst-case. A backtracking approach on suffix trees was proposed by Ukkonen [Ukk93], having look-up time $O(mq \log q + \mathrm{occ})$ and space requirement $O(mq)$ with $q = \min\{n, m^{d+1}\}$ for d errors. This was improved by Cobbs [Cob95] to look-up time $O(mq + \mathrm{occ})$ and space $O(q + n)$. No exact average-case analysis is available for these algorithms.

1.2.2 Registers for Approximate Text Indexing

Research has mainly concentrated on methods with a good practical or average-case performance. Until recently, the best known result for text indexing was due to Amir et al. [AKL$^+$00], who describe an index for one error under edit distance. The idea is based on building two suffix trees for the text and its reverse. A query for a pattern is implemented by computing for each prefix and each corresponding suffix of the pattern the intersection of the leaves in the subtrees of both suffix trees. For the latter, the leaves are labeled with the suffix (respectively, prefix) numbers and stored in two arrays. To avoid reporting matches multiple times, the character involved in the mismatch is added as a third dimension. On these arrays, a three-dimensional range searching algorithm is used to compute those leaves corresponding to substrings matching with one error. The size and preprocessing time of the range searching data structure is $O(n \log^2 n)$. Each range query takes time $O(\log n \log\log n)$, leading to an overall query time of $O(m \log n \log\log n + \mathrm{occ})$. By designing an improved algorithm for range searching over such tree cross products, this result was improved by Buchsbaum et al. [BGW00] to $O(n \log n)$ index preprocessing time and index size and $O(m \log\log n + \mathrm{occ})$ query time.

Another approach was taken by Nowak [Now04]. It solves the text indexing problem for one mismatch under Hamming distance with average preprocessing time $O(n \log^2 n)$, average index size $O(n \log n)$, and worst-case query time $O(m + \mathrm{occ})$. The algorithm is based on constructing a tree derived from the suffix tree of the text that contains all strings with one mismatch occurring directly after a node in the suffix tree. The error tree is constructed by successively merging subtrees of the suffix tree. Queries are implemented efficiently by making a case distinction upon the location of the mismatch. We extend the idea to edit distance allowing a constant number of errors and to other indexing problems in Chapter 5.

Recently, Cole et al. [CGL04] proposed a data structure allowing a constant number d of errors that works for don't cares (called wild cards), the Hamming distance, and the edit distance. For Hamming distance with d mismatches, a worst-case query

time of $O(m + (c \log n)^d/(d!) \cdot \log\log n + \text{occ})$ is achieved using a data structure that has size and preprocessing time $O(n(c \log n)^d/(d!))$ in the worst-case. For the edit distance, the query time becomes $O(m + (c \log n)^d/(d!) \cdot \log\log n + 3^d \text{occ})$, and the index size and preprocessing time is $O(n(c \log n)^d/(d!))$. The constant c varies for each given complexity. The idea is based on a centroid path decomposition of the suffix tree and the construction of errata trees for the paths. The complexities result from the fact that any string crosses at most $\log n$ centroid paths that have to be handled separately.

The approach of Cole et al. [CGL04] can also be used for the approximate dictionary indexing problem. For n strings of total size N, they achieve a query time of $O(m + (c \log n)^d/(d!) \cdot \log\log N + \text{occ})$ with $O(N + n(c \log n)^d/(d!))$ space and $O(N + n(c \log n)^d/(d!) + N \log N)$ preprocessing time for Hamming distance with d mismatches. For edit distance with d errors, their approach supports queries in time $O(m + (c \log n)^d/(d!) \cdot \log\log N + \text{occ})$ using and index that has preprocessing time $O(N + n(c \log n)^d/(d!) + N \log N)$ and size $O(N + n(c \log n)^d/(d!))$.

Previous results on dictionary indexing only allowed one error. For dictionaries of n strings of length m and Hamming distance, Yao and Yao [YY97] (earlier version in [YY95]) present and analyze a data structure that achieves a query time of $O(m \log\log n + \text{occ})$ and space $O(N \log m)$. The result is achieved by dividing the words recursively into two halves and searching one half exactly. To improve the query time, certain "bad queries" are handled separately. The analysis is conducted under a cell-probe model with bitwise complexity (i.e., counting bit probes), thus no preprocessing time is given.

Also for the Hamming distance and dictionaries of n strings of length m each (i.e., with $N = nm$), Brodal and Gąsieniec [BG96] present an algorithm based on tries, which uses similar ideas than those in [AKL$^+$00]. For the strings in the dictionary, two tries are generated, the second one containing the reversed strings. In the first trie, the strings are numbered lexicographically, while the other one is appended with sorted lists that are iteratively traversed upon a query and compared with the intervals from the first trie. The lists are internally linked from parent to child nodes, thus saving some work. Their data structure has size and preprocessing time $O(N)$ and supports queries with one mismatch in time $O(m)$.

Brodal and Venkatesh [BV00] analyze the problem in a cell-probe model with word size m. Translated to our uniform complexity, the query time of their approach is $O(m)$ using $O(N \log m)$ space. The idea is based on using perfect hash functions to quickly determine the position of an error. The data structure can be constructed in randomized expected time $O(Nm)$.

1.3 Thesis Organization

In Chapter 2, we introduce the relevant notation on strings, trees, and string distances. We describe concepts for the analysis of our index structures and algorithms and review some basic methods and data structures. We also give more details on the methods used for the average-case analysis.

Chapter 3 presents a linear time algorithm for bidirectional on-line construction of affix trees. We first describe the affix trees data structure. After reviewing algorithms

for the construction of suffix trees, we devise a linear-time on-line construction algorithm and analyze its complexity. The description of the necessary changes for the bidirectional construction and the amortized analysis finish the chapter.

The mathematically most challenging analysis is presented in Chapter 4. We compute the average complexity of an algorithm based on tries for approximate dictionary indexing and compare it to a simple approach that compares a pattern sequentially against the whole database. Therefore, we first give the analysis of the average complexity of the simple algorithm. We then derive an exact formula for the trie-based approach. After establishing some basic bounds, we compute the maximal expected speed-up. Then we analyze the complexity for a constant number of errors and finally compute the threshold on the number of errors where the performance of the trie-based approach becomes asymptotically the same as the performance of the naive method.

Chapter 5 presents a text index for approximate pattern matching with d errors. It has worst-case optimal query time and reasonable average size $O(n \log^d n)$ for d errors and text size n. We first formalize some intuitive properties of edit distance and introduce a slightly altered trie data structure. We then describe the main indexing structure based on error trees, detail the search algorithm, and prove size and time bounds. Next, we analyze the average size for an implementation with worst-case look-up time. The chapter finishes with the analysis of a modified approach that gives worst-case guarantees on the index size but only average-case bounds on the query time.

Finally, Chapter 6 summarizes our results and gives some directions for further research.

1.4 Publications

The results of this thesis have in part been published or announced at conferences. The results of Chapter 3 have appeared in [Maa00] and [Maa03]. The average-case analysis of the trie search algorithm in Chapter 4 has been published as an extended abstract [Maa04a] and as a technical report [Maa04b]. A journal version has been submitted to ALGORITHMICA. The results of Chapter 5 have been obtained in joint work with Johannes Nowak. In particular, the joint work included the idea of defining error trees recursively and making a distinction upon the position of the error and the height of the tree. The author of this work was responsible for basing the error trees on clearly defined error sets, which allowed for a straightforward generalization to any constant number of errors and enabled us to prove a useful dichotomy for the search algorithm. Furthermore, the author introduced range queries to select occurrences in subtrees, which solved the problem of answering queries in worst-case optimal time and allowed the adaption to a wide range of text indexing problems, and achieve a trade-off between query time and index size by defining weak tries. Additionally, the average-case analysis was provided by the author of this thesis. First results on an index for one error and edit distance have been submitted and accepted for publication [MN04]. An extended abstract for the full set of results will be presented at the CPM 2005 and is scheduled to appear in [MN05b]. A technical report describing the results has also been published [MN05a].

Chapter 2

Preliminaries

This work is concerned with the design and analysis of text indexes. In this chapter, we introduce the basic concepts and notation needed in order to describe our algorithms and conduct the analysis. Furthermore, we review some basic techniques related to text indexing.

2.1 Elementary Concepts

The meaning of text in a narrower sense is that of written text, e.g., in a book, which is structured into words. Text indexing refers to a broader definition of text including unstructured text such as DNA sequences. Therefore, we introduce the notion of a string. A string is a sequence of symbols. When creating an index on a string or a set of strings, we also refer to the latter as text or *text corpus*. We query an index with another string called the *pattern*.

We describe the basic notation for strings in the next section. We then introduce concepts of trees, which are an elementary search structure for strings. Finally, because real data is often far from being perfect, we conclude the section with some definitions of errors in strings.

2.1.1 Strings

A *string* is a sequence of symbols from an alphabet Σ. Unless otherwise stated, we assume a finite alphabet Σ. We let Σ^l denote the set of all strings of length l, i.e., the set of all character sequences $t_1t_2t_3\cdots t_l$ where $t_i \in \Sigma$. The special case Σ^0 is the set containing only the empty string denoted by ε. The set of all finite strings is defined as $\Sigma^* = \bigcup_{l\geq 0} \Sigma^l$ and the set of all non-empty, finite strings by $\Sigma^+ = \Sigma \setminus \{\varepsilon\}$. The *length* of a string is the number of characters of the sequence, i.e., the string $t = t_1t_2t_3\cdots t_l$ has length $|t| = l$. We can easily reverse a string $t = t_1t_2t_3\cdots t_l$, which we denote by $t^R = t_lt_{l-1}t_{l-2}\cdots t_1$. For strings $u, v \in \Sigma^*$, we denote their concatenation by uv. For $u, v, w \in \Sigma^*$ and $t = uvw$, u is a *prefix*, v a *substring*, and w a *suffix* of t. The substring starting at position i and ending at position j is denoted by $t_{i,j}$, where $t_{i,j} = \varepsilon$ whenever $j < i$. We denote by $t_{i,-}$ the suffix starting at position i and by $t_{-,j}$ the prefix ending at position j. We say that a string u *occurs* at position i of t if

$u = t_i \cdots t_{i+|u|-1}$. The *set of occurrences* of u in t is denoted by

$$\text{occurrences}_u(t) = \{i \mid u = t_i \cdots t_{i+|u|-1}\} \ . \tag{2.1}$$

A substring v is called *proper* if there exist $i \neq 1$ and $j \neq l$ such that $v = t_i \cdots t_j$. For a string $t = t_1 t_2 t_3 \cdots t_l$, the *set of substrings* is denoted by

$$\text{substrings}(t) = \{t_i \cdots t_j | 1 \leq i, j \leq l\} \ , \tag{2.2}$$

the *set of proper substrings* by

$$\text{psubstrings}(t) = \{t_i \cdots t_j | 1 < i \leq j < l\} \ , \tag{2.3}$$

the *set of suffixes* by

$$\text{suffixes}(t) = \{t_i \cdots t_l | 1 \leq i \leq l\} \ , \tag{2.4}$$

and the *set of prefixes* by

$$\text{prefixes}(t) = \{t_1 \cdots t_i | 1 \leq i \leq l\} \ . \tag{2.5}$$

A suffix u of t is called *proper* if it is not the same as t, i.e., $u \neq t$. It is called *nested* if it has at least two occurrences, i.e., additionally to $u \in \text{suffixes}(t)$ we have $u \in \text{psubstrings}(t) \cup \text{prefixes}(t)$. Alternatively defined, a suffix u of t is nested if $u \in \text{suffixes}(t)$ and $|\text{occurrences}_u(t)| > 1$. Similarly, a prefix u of t is called proper if it is not the same as t, and it is called nested if it is a proper prefix with more than one occurrence in t. We call a substring u of t *right-branching* if it occurs at two different positions i and j of t followed by different characters $a, b \in \Sigma$, i.e., $t_{i,i+|u|} = ua$, $t_{j,j+|u|} = ub$, and $a \neq b$. A substring u of t is called *left-branching* if it occurs at two different positions i, j of t preceded by different characters $a, b \in \Sigma$, i.e., $t_{i-1,i+|u|-1} = au$, $t_{j-1,j+|u|-1} = bu$, and $a \neq b$.

We illustrate the previous definitions with the string `banana`: The suffix `banana` is not proper, the suffix `anana` is proper but not nested, and the suffix `ana` is proper and nested. There is no right-branching substring in `banana`, but the substring `ana` is left-branching because `bana` and `nana` are substrings of `banana`. The following important fact can be easily seen.

Observation 2.1 (The right-branching property is inherited by suffixes).
If w is right-branching in t, then all suffixes of w are also right-branching in t.

Let $S \subseteq \Sigma^*$ be a finite set of strings. The size of S is defined as $\|S\| = \sum_{u \in S} |u|$. The cardinality of S is denoted by $|S|$. For any string $u \in \Sigma^*$, we let $u \in \text{prefixes}(S)$ denote that there is a string v in S such that $u \in \text{prefixes}(v)$. We define $\text{maxpref}(S)$ to be the length $|u|$ of the longest common prefix u of any two different strings in S. The *prefix-free reduction* of a set of strings S is the set of strings in S that have no extension in S, i.e., those strings that are not a prefix of any other string. We denote the prefix-free reduction of S by

$$\text{pfree}(S) = \{u \mid u \in S \text{ and } \forall a \in \Sigma ua \notin S\} \ . \tag{2.6}$$

A set S is called *prefix-free* if no string is the prefix of any other string, that is, $\text{pfree}(S) = S$.

2.1.2 Trees over Strings

Trees are used in the implementation of many well-known search structures and algorithms, e.g., AVL-trees, (a,b)-trees, red-black trees [Knu98, CLR90]. When the universe of keys consists of strings, we can do better than the aforementioned comparison-based methods. This is called "digital searching" [Knu98]. The basic idea is captured in the definition of Σ^+-*trees* [GK97]. A Σ^+-tree $\mathcal{T}$ is a rooted, directed tree with edge labels from Σ^+. For each $a \in \Sigma$, every node in $\mathcal{T}$ has at most one outgoing edge whose label starts with a (*unique branching criterion*). In essence, a Σ^+-tree allows a one-to-one mapping between certain strings and the nodes of the tree. An edge is called *atomic* if it is labeled with a single character. A Σ^+-tree is called atomic if all edges are atomic. A Σ^+-tree is called *compact* if all nodes other than the root are either leaves or branching nodes. Replacing sequences of nodes that each have only one child by a single edge with the concatenated labels is called *path compression*.

Let p be a node of the Σ^+-tree $\mathcal{T}$, we denote this by $p \in \mathcal{T}$. Let u be the string that results from concatenating the edge labels on the path from the root to p. We define the *path* of p by $\mathrm{path}(p) = u$. By the unique branching criterion there is a one-to-one correspondence between u and p. In such a case, it is convenient to introduce the notation $\overline{u}$ to identify the node p. We define the *string depth* of a node p to be $\mathrm{depth}(p) = |\mathrm{path}(p)|$. The node depth is defined as the number of nodes on the path from the root to a node. The latter is less important to us because—since edge labels are never empty—it is bounded from above by the string depth. In the following, we thus refer to the string depth simply by the *depth*. For a Σ^+-tree, we define its *height* to be the maximal depth of an inner node.

Each Σ^+-tree $\mathcal{T}$ defines a *word set* that is represented by the tree. The word set $\mathrm{words}(\mathcal{T})$ is the set of strings that can be read along paths starting at the root of the tree. Formally, we have

$$\mathrm{words}(\mathcal{T}) = \{u \mid \text{there exists } p \in \mathcal{T} \text{ with } u \in \mathrm{prefixes}(\mathrm{path}(p))\} \quad . \tag{2.7}$$

Note that $\mathrm{pfree}(\mathrm{words}(\mathcal{T}))$ is exactly the set of strings represented by the leaves of $\mathcal{T}$.

For an atomic or a compact Σ^+-tree $\mathcal{T}$ there is a one-to-one correspondence between the set of strings $\mathrm{words}(\mathcal{T})$ and the tree $\mathcal{T}$. For an atomic Σ^+-tree $\mathcal{T}$ there is even a node p with $\mathrm{path}(p) = u$ for every string $u \in \mathrm{words}(\mathcal{T})$. For non-atomic, especially for compact Σ^+-trees, we distinguish whether a string has an *explicit* or an *implicit location*. If for a string $u \in \mathrm{words}(\mathcal{T})$ there exists a node $p \in \mathcal{T}$, then we say that u has an explicit location, otherwise we say that u has an implicit location. Whereas explicit locations can conveniently be represented by a node, we have to introduce *virtual nodes* for implicit locations. Let $u \in \mathrm{words}(\mathcal{T})$ be a string with an implicit location. By definition, there must be a node p with $u \in \mathrm{prefixes}(\mathrm{path}(p))$. Let q be the parent of p, then $u = \mathrm{path}(q)v$ for some $v \in \Sigma^+$. We define the pair (q, v) to be the virtual node for u. A virtual node is a concept that allows us to treat compact Σ^+-trees in the same way as atomic Σ^+-trees. With a proper representation of a virtual node (e.g., as a (canonical) reference pair in Chapter 3), we can extend this to an algorithmic level—designing algorithms for atomic Σ^+-trees but representing them more space efficiently by compact Σ^+-trees.

For the Σ^+-tree $\mathcal{T}$, we denote by $\mathcal{T}_p$ the subtree rooted at node p. It makes no difference whether p is a real or virtual node. It is convenient to use the same notation

for strings. For $u \in \mathrm{words}(\mathcal{T})$, we let $\mathcal{T}_u$ denote the subtree rooted at the (possibly virtual) node p with $\mathrm{path}(p) = u$. The words represented by $\mathcal{T}_u$ are special suffixes of the word set represented by $\mathcal{T}$, i.e., $\mathrm{words}(\mathcal{T}_u) = \{v \mid uv \in \mathrm{words}(\mathcal{T})\}$.

2.1.3 String Distances

Wherever real data is entered into a computing system either by hand or by some automatic device, there is the chance for errors. Furthermore, many times the process of generating the data, e.g., by measuring it, may be error prone. Therefore, it is necessary to describe and handle errors in texts and strings. There are many possible ways of describing the distance (or similarity) of two strings. A very general model of distance can be described by a set of rules called *operations* of the type $\Sigma^* \to \Sigma^*$ associated with a certain cost. The distance is the total cost needed to transform one string into the other by sequentially applying the operations to substrings. Because this general model allows encoding a Turing machine computation, the distance between two strings may be undecidable. In this work, we only deal with distances based on operations of the type $\Sigma \cup \{\varepsilon\} \to \Sigma \cup \{\varepsilon\}$, which are easily computable by dynamic programming. It is possible to define distances that operate on larger blocks, but there is no generally accepted, widespread model. Furthermore, if the model is not carefully chosen, it might not be easily computable, e.g., there are block distances that are NP-complete [LT97].

The restriction to operations over substrings of at most one character results in three different *edit operations*: *insertion* $\{\varepsilon\} \to \Sigma$, *deletion* $\Sigma \to \{\varepsilon\}$, and *substitution* $\Sigma \to \Sigma$. The cost of each operation can be chosen differently by the type of the operation and by the characters that are involved. It is also possible to restrict the use only to certain types of operations, which can also be achieved by setting the cost of the undesired operations to infinity. We assume that the so defined distances are positive, symmetric, and obey the triangle inequality to deserve their name.

In the simplest version, we only allow substitutions. This corresponds to comparing two strings character-wise. For these *comparison-based* string distances, we assign a value $\mathrm{d}(a, b) \in \{0, 1\}$ to the comparison of each pair of characters $a, b \in \Sigma$. Thus, the distance between two strings $u, v \in \Sigma^*$ is simply defined by

$$\mathrm{d}(u, v) = \begin{cases} \infty & \text{if } |u| \neq |v|, \\ \sum_{i=1}^{|u|} \mathrm{d}(u_i, v_i) & \text{otherwise.} \end{cases} \tag{2.8}$$

The most elementary comparison-based distance is the *Hamming distance* where $\mathrm{d}(a, b)$ is zero if and only if $a = b$, and it is one otherwise [Ham50]. The Hamming distance counts the number of mismatches between two strings of the same length. A common extension are *don't-care symbols*. A don't-care symbol x is a special character for which $\mathrm{d}(x, a)$ is always zero.

If we allow only insertions and deletions, we get the *longest common subsequence* distance. The value is often divided by the string length to yield a value between zero and one, giving some measure of string similarity.

The general case is usually called *weighted edit distance*. It can be computed via dynamic programming using the following recursive equation. For $u \in \Sigma^k$ and $v \in \Sigma^l$,

we compute

$$\mathrm{d}\left(u_1\cdots u_k, v_1\cdots v_l\right) = \min \left\{ \begin{array}{l} \mathrm{d}\left(u_1\cdots u_k, v_1\cdots v_{l-1}\right) + \mathrm{d}\left(\varepsilon, v_l\right) \\ \mathrm{d}\left(u_1\cdots u_{k-1}, v_1\cdots v_l\right) + \mathrm{d}\left(u_k, \varepsilon\right) \\ \mathrm{d}\left(u_1\cdots u_{k-1}, v_1\cdots v_{l-1}\right) + \mathrm{d}\left(u_k, v_l\right) \end{array} \right\} , \quad (2.9)$$

where $\mathrm{d}\left(\varepsilon, \varepsilon\right) = 0$. In the above formula, the top line represents an insertion, the middle line a deletion, and the bottom a substitution or a match. In the most basic variant, we assign a cost of one for an insertion, deletion, or substitution and a cost of zero for a match. The result is simply called *edit distance* or Levenshtein distance [Lev65]. It counts the minimal number of insertions, deletions, and substitutions to transform one string into another.

We should mention that there are many other models for string distance but none as thoroughly studied as the aforementioned ones. Besides those also considered in the pattern matching or computational biology community (see [Nav01] for a list and references), there are a number of specialized distances in information retrieval or record linkage (see, e.g., [PW99]).

2.2 Basic Principles of Algorithm Analysis

Before embarking on the design and analysis of our algorithms, we review some basic principles. In particular, we have to agree on how to measure the cost or complexity of an algorithm and a data structure, i.e., we describe the measure for our resources time and space. Secondly, we review amortized analysis, which is used heavily in Chapter 3. Finally we describe the basic concepts of average-case analysis, needed for Chapters 4 and 5.

2.2.1 Complexity Measures

In order to assess the quality of an algorithm by comparing it with other algorithms, we need a common measure. The basic resources we compare are time and space usage. We compare these asymptotically by describing the relation between the size of the input, and the space and time needed. The *complexity* of an algorithm is thus a description of the growth in the amount of resources used.

There are basically two choices for time complexity and two for space complexity. Most computers do not work with bits but with chunks of bits of a certain size, called words. Basic operations are usually implemented in the main processor to take only a constant number of clock ticks (e.g., addition, comparison, or even multiplication). The word size usually also affects the memory access, i.e., the bus width. Therefore, a word is often assumed as a basic unit in the complexity analysis.

Regarding *time complexity*, we can either assume a *uniform cost model* with constant time per operation or a *logarithmic cost model* where the cost of each operation is linear in the number of bits of the numbers operated on. The uniform cost model is very popular because it reflects the situation where the size of numbers and pointers does not exceed the word size of the computer. This is usually the case for text algorithms when all data fits into the main memory, for which a pointer can normally be stored in a single word. Usually, even the size of the data storable on a random

access hard disk does not exceed one or two words. While operations like multiplication can indeed depend on the size of the number, this fact is often not visible due to advanced microprocessor architecture features like pipelining, out-of-order execution, or caching strategies (memory access is often a bottleneck).

Especially regarding text indexes, there are two main models for measuring *space*. The *word model* counts the number of words of memory used. A more detailed analysis, called *bit model*, accounts the number of bits used. When the size of the input is limited, e.g., by the main memory, the difference in both models is only a constant factor (the word size). In practice, the size of an index plays an important role, so the constant factor hidden by the Landau notation is of interest.

Unless explicitly stated otherwise, we assume a uniform cost model for time complexity and the word model for space complexity in this work.

2.2.2 Amortized Analysis

Amortized analysis is a technique to bound the worst-case time complexity of an algorithm. It is commonly used to analyze a sequence of operations and prove a worst-case bound on the average time per operation. There are three basic techniques called aggregate, accounting, and potential method (see, e.g., Chapter 18 in [CLR90]). We make heavy use of the accounting method in Chapter 3; therefore, we review the method on dynamic tables, a data structure that we also need in Chapter 3.

A *dynamic table* is a growing array with amortized constant cost for adding an element. Elements are added to the end until a previously reserved space is filled. Each time this happens, a new, larger space is claimed, and the old elements are copied to this space. Referencing and changing elements works in the same way as for normal arrays. Dynamic tables can also be implemented to support deletions, but we refer the reader to [CLR90] for the details.

The basic idea of amortized analysis is to distribute the cost arising for certain operations, e.g., when the array reaches its current maximum and the elements need to be copied to a new location. Therefore, we consider an infinite sequence of operations and show that the first n operations can be executed in n steps, i.e., every step takes *amortized constant time*. By the aggregate technique, we would just bound the total number of operations and divide it by n. By the potential method, we would assign a certain potential to the data structure (much like the potential energy of a bike on top of a hill) and "use" it to perform certain costly operations that reduce the potential. Here, for the accounting method, we assign coins to certain places in the data structure. In particular, we put two coins on each added element to pay for the subsequent restructuring. In a sense, we put a certain amount of time in a bank account to be used later. The accounting method differs from the potential method in that one cannot necessarily "see" the coins when looking at the data structure. Therefore, we have to prove that the coins are available when we need them.

For the dynamic table, we only describe the data structure and the operation to add an element. The data structure consists of an array A of size m and a fill level l. It stores l elements. As long as $l+1 \leq m$, a new element is stored in $A[l+1]$. When the fill level reaches m, a new array A' of size $2m$ is created and the elements from A are copied to A'. The array A is then released and replaced by A'. We need to perform m additional operations to copy the elements from A to A'. Afterwards, we can store the

new element in $l+1$ as usual. Note that after we have copied elements from A to A' we have $l = \frac{m}{2}$.

To prove that adding an element takes amortized constant time, we add two coins on each newly inserted element if $l+1 \leq m$. Thus, the amortized time to add an element is $2+1 = 3$ operations for adding two coins and storing the new element in A. At the time where the array has to be enlarged, we have added the elements $A[\frac{m}{2}+1]$ to $A[m]$. On each of these elements, we have stored two coins. Therefore, we have exactly m coins, which we use to pay for the time needed to copy the elements from A to the newly allocated A'. We then add the new element together with two coins. As a result, adding a new element has an amortized constant cost of one real and two delayed operations.

2.2.3 Average-Case Analysis

Average-case analysis analyzes the expected behavior of algorithms. This is especially interesting in the case that an algorithm is executed many times with different inputs, thus giving Fortuna a chance to do her duty and even out some inputs with high costs. The primary goal of average-case analysis is to find a better understanding of the behavior of algorithms as it is observed in practice. The usefulness of an average-case analysis also depends upon the adopted probabilistic model that has to be sufficiently realistic. The model is therefore the basis of the analysis. In this section, we describe some common models for average-case analysis of string algorithms. A very thorough treatment of the subject is available in [Szp00].

In the average-case analysis of string algorithms, the random input is usually a string or a set of strings. It is common to assume that each string w is the prefix of an infinite sequence of symbols $\{X_k\}_{k\geq 1}$ generated by a random source from the alphabet Σ. The assumptions about the source determine the probabilistic model.

The simplest source is the *memoryless source*. The sequence $\{X_k\}_{k\geq 1}$ is assumed to be the result of independent Bernoulli trials with a fixed probability $\Pr\{X_k = a\}$ for each character $a \in \Sigma$. We distinguish the asymmetric case where the probability for each character may be different and the symmetric case where $\Pr\{X_k = a\} = \frac{1}{|\Sigma|}$.

When the assumption of character-wise independence is too strong, one can use a *Markov source*. Here, the probability of each character depends upon the history, i.e., it depends on the l previous characters for an l-th order Markov chain. Usually, only first order Markov chains are treated, where the probabilities are given by $\Pr\{X_k = a | X_k = b\}$. We do not work with Markov chains but with the following more general model.

Memoryless and Markov sources are comprised in *stationary and ergodic sources*. An in-depth treatment can be found in [Kal02]. Let $(\Omega, \mathfrak{F}, \mu)$ be a probability space for the random sequence $\{X_k\}_{k\geq 1}$. The set Ω contains all possible realizations of strings, that is,

$$\Omega = \{\omega; |\ \omega = \{\omega_k\}_{k\geq 1} \text{ and } \omega_k \in \Sigma \text{ for all } k\} \ , \tag{2.10}$$

the σ-field $\mathfrak{F}$ is generated by all finite sets

$$\mathbb{F}_1^m = \{\omega_1 \cdots \omega_m; |\ \omega_k \in \Sigma \text{ for all } 1 \leq k \leq m\} \ , \tag{2.11}$$

and μ is a probability measure on $\mathfrak{F}$. Let $\theta : \Omega \to \Omega$ be the shift operator, i.e.,

$$\theta(\omega_1, \omega_2, \omega_3, \ldots) = \omega_2, \omega_3, \ldots \quad . \tag{2.12}$$

A random sequence $\{X_k\}_{k\geq 1}$ is called *stationary* if $\theta(\{X_k\}_{k\geq 1})$ has the same distribution as the sequence itself, that is, θ is measure preserving: $\mu \circ \theta^{-1} = \mu$.

A set $I \subset \Omega$ is called shift-invariant if $\theta^{-1}(I) = I$. Let $\mathcal{S}$ be the set of shift-invariant sets in Ω, then $\mathcal{S}$ is a σ-field. A stationary sequence is called *ergodic* if the shift-invariant σ-field is trivial, i.e., for each element $I \in \mathcal{S}$ either $\mu(I) = 0$ or $\mu(I) = 1$. Stationary and ergodic sequence are of interest because the time and space means are equal, i.e., we can find the expected value of the X_k by averaging over the sequence: $\mathbb{E}[X_k] = n^{-1}\sum_{i=1}^{n} X_i$.

Once we have fixed the probabilistic model, we can analyze the expected time or space complexity using various methods from probability theory and also analytical tools (an approach already taken in [Knu98]). Before we embark on this tasks, we introduce some additional properties needed and related results.

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

$$c_1 \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \leq \Pr\{\mathcal{A} \cap \mathcal{B}\} \leq c_2 \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \tag{2.13}$$

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

$$r_2 = \lim_{n\to\infty} \frac{-\ln\left(\sum_{w\in\Sigma^n} (\Pr\{w\})^2\right)}{2n} \quad , \tag{2.14}$$

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

The source satisfies the *strong α-mixing condition* if a function $\alpha : \mathbb{N} \to \mathbb{R}$ exists with $\lim_{d\to\infty} \alpha(d) = 0$ such that, for all $1 \leq m \leq m + d \leq n$ and for all $\mathcal{A} \in \mathbb{F}_1^m$, $\mathcal{B} \in \mathbb{F}_{m+d}^n$, the inequality

$$(1 - \alpha(d)) \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \leq \Pr\{\mathcal{A} \cap \mathcal{B}\} \leq (1 - \alpha(d)) \Pr\{\mathcal{A}\} \Pr\{\mathcal{B}\} \tag{2.15}$$

holds.

We do not only consider expected complexities in the average-case analysis of algorithms or data structures, but also probabilistic bounds, i.e., statements about the probability that certain events occur. In particular, a behavior is witnessed *with high probability* (w.h.p.) when the probability grows as $1 - o(1)$ with respect to the input size n, i.e., the probability approaches one as n increases. For random variables depending on growing sequences, we can also define convergence probabilities. A random variable X_n converges *in probability* to a value x if for any $\epsilon > 0$

$$\lim_{n\to\infty} \Pr\{|X_n - x| < \epsilon\} = 1 \quad . \tag{2.16}$$

A random variable X_n converges *almost surely* to a value x if for any $\epsilon > 0$

$$\lim_{N \to \infty} \Pr \left\{ \sup_{n \geq N} |X_n - x| < \epsilon \right\} = 1 \ . \tag{2.17}$$

As an example and for later use, we reproduce some results regarding the *maximal length of a repeated substring* in a string of length n. We recapitulate here results from [Szp93a] (see also [Szp93b]). Let H_n denote the maximal length of a repeated string with both occurrences starting in the first n characters of a stationary and ergodic sequence $\{X_k\}_{k \geq 1}$. If the source satisfies the mixing condition (2.13) and if $r_2 > 0$, then one can prove that

$$\Pr \left\{ H_n \geq (1 + \epsilon) \frac{\ln n}{r_2} \right\} \leq c n^{-\epsilon} \ln n \ , \tag{2.18}$$

for some constant c and $\epsilon > 0$. Thus, H_n is with high probability $O(\ln n)$. Indeed, even almost sure convergence can be proven:

$$\lim_{n \to \infty} \frac{H_n}{\ln n} = \frac{1}{r_2} \qquad \text{(a.s.)} \ , \tag{2.19}$$

which is valid if the source satisfies the strong α-mixing condition and additionally $\sum_{d=0}^{\infty} \alpha(d) < \infty$ holds. The latter condition is always fulfilled for stationary and ergodic Markov chains [Bra86]. We have just bounded the (string) height of a suffix tree that we will introduce in Section 2.3.1. Most Σ^+-trees built from sets of random string generated by a stationary and ergodic source have a logarithmic height [Pit85].

2.3 Basic Data Structures

We already described dynamic tables in Section 2.2.2. In Chapter 4, we analyze a search algorithm on tries or PATRICIA trees. The index structure described in Chapter 5 works for all kinds of Σ^+-trees, e.g., tries, PATRICIA trees, suffix trees, and generalized suffix trees. We introduce all these data structures in the following section. Section 2.3.2 describes how to implement range queries in optimal time, which we also need in Chapter 5.

2.3.1 Tree-Based Data Structures for Text Indexing

In this section, we present some of the most common data structures for text indexing in terms of the Σ^+-tree defined above. These structures are the basis of our algorithms. We describe here how the data structures are interpreted theoretically and also give some basic idea on how they can be implemented.

There are many implementation aspects for Σ^+-trees depending on the application. Some applications require that the parent of a node is accessible in constant time whereas others always traverse the tree from the root to the leaves. Some basic techniques can be identified.

In many cases, the edges are not labeled explicitly but by using pointers into an underlying string or set of strings. This can be either a start and an end pointer or a start

pointer and a length. In this way, each edge takes constant space while still allowing constant time access to the label. Furthermore, it may be possible to store the pointers implicitly (see, e.g., [GKS03]). Because we are dealing with trees, it is also common to store the edges (i.e., their parameters) at the target nodes.

Another choice concerning the implementation is the organization of branching. The children of a node are commonly accessed by the first character of the corresponding edge label. The most common techniques are arrays, hashing, linked list, search trees, or a combination of these. The fastest access is possible using arrays indexed by the first character of the corresponding edge label, which takes $|\Sigma|$ pointers per node. The most space efficient implementation uses a linked list, which needs one pointer per child. Here the access time is proportional to the number of children or $O(|\Sigma|)$ in the worst case. Hashing and search trees provide solutions which compromise between both extremes. Asymptotically, the solutions are all equivalent because we assume a constant size alphabet.

The simplest variant of a Σ^+-tree is the *trie*, introduced by Brandais [Bri59] and by Fredkin [Fre60]. A trie for a prefix-free set of strings S is a Σ^+-tree $\mathcal{T}$ such that $\mathrm{pfree}(\mathrm{words}(\mathcal{T})) = S$, i.e., each string in S is represented by a leaf, and there are no other leaves in $\mathcal{T}$. The representation of edges does not matter asymptotically, but the classic trie has atomic edges from the root to the inner nodes and then compactifies the edges leading to the leaves. An example is given in Figures 2.1(a) and 2.1(b).

The idea of compressing all paths of a trie leads to the *PATRICIA tree* introduced by Morrison [Mor68]. An example is shown in Figure 2.1(c). A PATRICIA tree is essentially a simple representation of the compact Σ^+-tree for the set of strings S. It is not uncommon to talk of *compact tries* instead of PATRICIA trees.

A *suffix tree* for a string t is a Σ^+-tree that contains all the substrings of t. Formally, a suffix tree T for a string t must satisfy

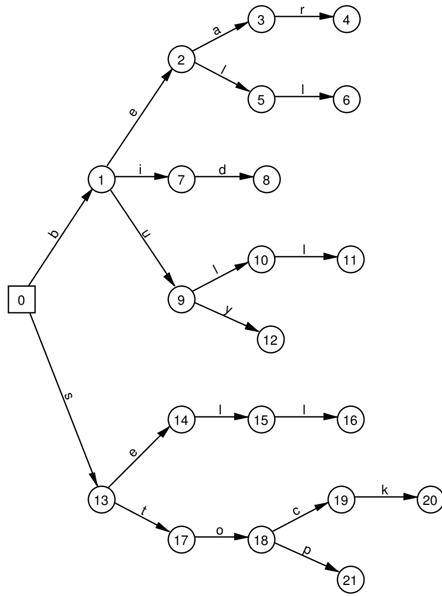
$$\mathrm{words}\,(\mathcal{T}) = \mathrm{substrings}\,(t) \quad . \tag{2.20}$$

Usually, one is only interested in the compact suffix tree (*CST*) because it has linear size. In the following we will always refer to the compact suffix tree simply by the suffix tree.

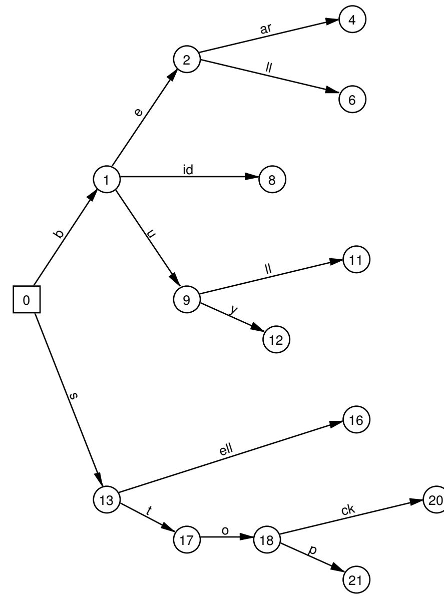
The example of a suffix tree in Figure 2.1(d) also includes additional edges (dashed arrows). These are called *suffix links*. A suffix link points from a node p to a node q if $\mathrm{path}(q)$ represents a suffix of $\mathrm{path}(p)$ in $\mathcal{T}$. In particular, $\mathrm{path}(q)$ should be the longest proper suffix of $\mathrm{path}(p)$. In suffix trees, usually only suffix links for inner nodes are added. Historically, suffix links were an invention to facilitate linear-time construction of suffix trees [Wei73, McC76, Ukk95], but there exist other applications for suffix links in suffix trees, e.g., computing matching statistics [CL94] or finding tandem repeats in linear time [GS04].

Another common feature of suffix trees visible in Figure 2.1(d) is the use of a *sentinel*, in this case the character '`$`'. A sentinel is a character that does not occur anywhere else in the string. By the addition of such a unique character, there can be no nested suffixes, thus each suffix is represented by a leaf in the suffix tree.

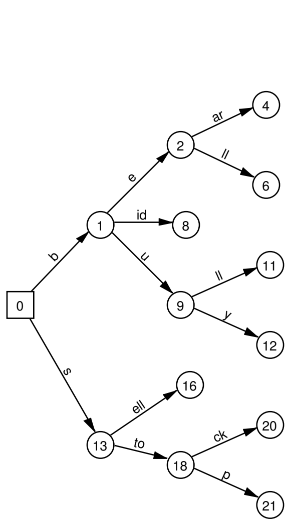
If the leaves are labeled by the number of the suffix of t that they represent, then the suffix tree can be used to find all occurrences of a pattern w. Starting from the root, we simply follow the edges which spell out w. The leaves of the subtree $\mathcal{T}_w$ represent



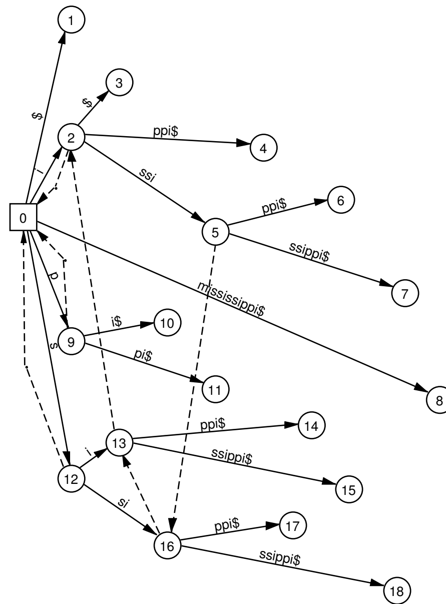
(a) A trie



(b) A trie with compressed leaf edges



(c) A PATRICIA tree



(d) A suffix tree

Figure 2.1: Examples of Σ^+-trees representing the prefix-free set of strings `bear`, `bell`, `bid`, `bull`, `buy`, `sell`, `stock`, `stop` (a-c) and a suffix tree for `mississippi$` (d).

the occurrences of w, i.e., if the leaf $p \in \mathcal{T}_w$ is labeled i, then there is an occurrence of w starting at i. Hence, we can report l occurrences of a pattern w in a text t in time $O(|w| + l)$. The suffix tree thus solves the *text indexing problem* in optimal time. The problem is to preprocess a given text t to answer the following queries: Given a pattern w, report all i such that $t_{i,i+|w|-1}$ is equal to w.

As noted already, suffix trees can be constructed in linear time [Wei73, McC76, Ukk95, Far97] (Ukkonen's algorithm is described in Chapter 3). It is also possible to build a suffix tree for a set of strings, a so-called *generalized suffix tree*. A Σ^+-tree $\mathcal{T}$ is a generalized suffix tree for the set of strings S if

$$\text{words}\,(\mathcal{T}) = \bigcup_{t \in S} \text{substrings}\,(t) \quad . \tag{2.21}$$

The easiest way to construct a generalized suffix tree for a set S is to create a new string $u = t_1\$_1 t_2\$_2 \cdots t_{|S|}\$_{|\mathtt{S}|}$ as the concatenation of all strings t_i in S separated by different sentinels $\$_\mathtt{i}$. The suffix tree for u is equivalent to the generalized suffix tree for S if leaves are cut off after each sentinel. Note that it is not necessary to use $|S|$ different sentinels. We can use a comparison function that considers two sentinels to be different. The drawback is that there might be two outgoing edges labeled by the sentinel at a single node which must then be handled in matching and retrieval.

To conclude this section, we mention that there are alternative data structures such as the directed acyclic word graph (DAWG) [BBH+87], the suffix array [MM93], the suffix cactus [Kär95], or the suffix vector [MZS02]. The most important of these is the suffix array, which has become a viable replacement for the suffix tree [AKO04].

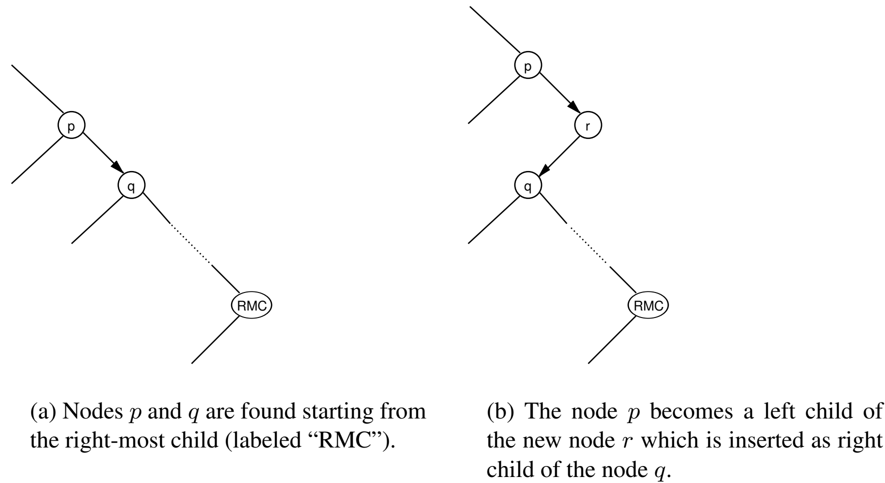
2.3.2 Range Queries

In the previous section, we described a method to find all l occurrences of a pattern w in a string t by matching the pattern in the suffix tree built from t and reporting the indices stored at the leaves as output. This algorithm runs in time $O(|w| + l)$ which is linear in the size of the query pattern and the number of reported indices. We call such a behavior *output sensitive*. An algorithm is output sensitive if it is optimal with respect to the input and output, i.e., it has a running time that is linear in the size of the input and of the outputs.

Another important problem is the *document listing problem*. The problem is to preprocess a given set of strings $S = \{d_1, \ldots, d_m\}$ (called documents) to answer the following queries: Given a pattern w, report all i such that d_i contains w.

Let $\mathcal{T}$ be a generalized suffix tree for S where the leaves are labeled with the document numbers, i.e., leaf p is labeled with k if $\text{path}(p)$ is a suffix of d_k. The document listing problem can be solved by matching the pattern w and reporting all different document numbers occurring at leaves in $\mathcal{T}_w$. Unfortunately, the number of leaves may be much larger than the number of reported documents. The simple algorithm is not output sensitive. This and other similar problems can be solved optimally by using constant time range queries [Mut02], which we present next.

The following range queries all operate on arrays containing integers. The goal is to preprocess these arrays in linear time so that the queries can be answered in constant time.



(a) Nodes p and q are found starting from the right-most child (labeled "RMC").

(b) The node p becomes a left child of the new node r which is inserted as right child of the node q.

Figure 2.2: One step in the linear-time algorithm to build the Cartesian tree. After inserting the node r, it becomes the new right-most child. Observe how no node on the path from the old right-most child to the node p can ever be on the right-most path again.

A *range minimum query* (RM) for the range (i, j) on an array A asks for the smallest index containing the minimum value in the given range. Formally, we seek the smallest index k with $i \leq k \leq j$ and $A[k] = \min_{i \leq l \leq j} A[l]$.

A *bounded value range query* (BVR) for the range (i, j) with bound b on array A asks for the set of indices L in the given range where A contains values less than or equal to b. The set is given by $L = \{l \mid i \leq l \leq j \text{ and } A[l] \leq b\}$.

A *colored range query* (CR) for the range (i, j) on array A asks for the set of distinct elements in the given range of A. These are given by $C = \{A[k] \mid i \leq k \leq j\}$.

RM queries can be answered in constant time and with linear processing [GBT84]. The idea is based on lowest common ancestor (LCA) queries, which can be answered in constant time [HT84, SV88, BFC00, BFCP$^+$01]. We briefly describe an algorithm for RM queries along the lines of [BFC00].

The first step is to build the *Cartesian tree* $\mathcal{B}$ for the array A. Let A be indexed 1 through n. The Cartesian tree [Vui80] is a binary tree. The root represents the smallest index i with the minimum element in A, the left subtree is a Cartesian tree for the subarray $A[1 \ldots i-1]$, and the right subtree is a Cartesian tree for the subarray $A[n \ldots i-1]$. The Cartesian tree $\mathcal{B}$ can be built in linear time by iteratively building the tree for the subarray $A[1 \ldots i]$. At each step, we keep a pointer to the right-most node (the node we reach when taking only right edges). When the new value $A[i]$ is added, we move towards the root until we find a node p containing a smaller value. If no such node is found, the new value is the smallest so far and becomes the new root with the old root being its left child. Otherwise, let q be the right child of p (if it exists). We add a new node r with the value $A[i]$ in place of q as the right child of p and make q the left child of r. Figure 2.2 shows a sketch of this step. Because every traversed node ends up in the left subtree of r, no node is traversed twice in this way. As a result, the total time to build $\mathcal{B}$ is $O(n)$.

In the second step, we construct three new arrays from an Euler tour of the Cartesian tree. The first array H of size $2n$ contains the node depths encountered during the traversal, the second array E of size $2n$ contains the indices of A stored at the nodes, and the third array M of size n stores an occurrence in E for each index in A. The Cartesian tree can be discarded after this step. Note that successive values in H differ by plus or minus one, thus we call H a ± 1-array. Note further that, if k is the index of the minimum element in the range (i, j) of H, then $A[E[k]]$ is the minimum element in the range $(E[i], E[j])$ of A. This allows answering RM queries on A by RM queries on H.

As a last step, we need to prepare RM queries on the ± 1-array H. We divide H into blocks of length $b = \lceil \frac{\log n}{2} \rceil$. A doubly indexed array L contains in $L[i, j]$ the index of the minimum value in $H[ib \ldots (i + 2^j - 1)b]$. It has size $O(n)$ and we can construct it by dynamic programming using the equation

$$L[i,j] = \begin{cases} L[i, j-1] & \text{if } H[L[i, j-1]] < H[L[i + 2^{j-1} - 1, j-1]] \\ L[i + 2^{j-1} - 1, j-1] & \text{otherwise.} \end{cases} \tag{2.22}$$

The values $L[i, 1]$ are computed by a simple scan of H. Because H is a ± 1-array, we represent each block of size b by an integer x between zero and $2^b \leq 2\sqrt{n}$. For each pair of relative offsets (k, l) we store in an array $B[x, k, l]$ the relative offset of the smallest value. We have $0 \leq k, l < b$, thus B has size $O(\sqrt{n} \log^2 n)$. An additional array P of size $\lceil \frac{n}{b} \rceil$ is used to store the integer representation of the blocks.

To find the index of the minimum element in H in the range (i, j) we first compute the block boundaries $i' = \lceil \frac{i}{b} \rceil$ and $j' = \lfloor \frac{j}{b} \rfloor$. If we have $j - i < b$, then we can find the index of the minimum element in H by looking up $B[P[i'], i - bi', j - bi']$. Otherwise we use array L to find the index of the minimum element in the range (bi', bj') by comparing $H[L[i', k]]$ and $H[L[j' + 2^{k-1} - 1, k]]$ where k is chosen such that $2^k \leq i' - j' \leq 2^{k+1}$. For the ranges (i, bi') and (bj', j), we look up the indices of the minima in the array B by $B[P[i'], i - bi', b - 1]$ and $B[P[j' + 1], 0, j - bj']$. As a result, we can answer minimum range queries on A in constant time with the arrays H, E, M, L, B, P using $O(n)$ space.

BVR queries on array A can be answered based on RM queries [Mut02] in time $O(|L|)$ and with linear preprocessing. The idea is to successively perform RM queries: Let k be the answer to the RM query (i, j), then either $A[k] > b$, in which case we are finished with this subrange, or $A[k] \leq b$, in which case we continue by querying $(i, k-1)$ and $(k+1, j)$. Each query either stops or yields a new element of L. Thus, we get a binary tree like query structure, the inner nodes corresponding to successful queries and the leaves to unsuccessful queries or empty intervals. The tree is binary, so we perform $O(|L|)$ queries in total.

Finally, CR queries can also be answered in time $O(|C|)$ and with linear preprocessing [Mut02]. The solution is based on BVR queries. A new array R is constructed with $R[i] = j$ if $j < i$ is the largest index with $A[j] = A[i]$, and with $R[i] = -1$ if no such index exists. Thus, a colored range query (i, j) on A can be answered by a bounded value range query (i, j) with bound i on R: The resulting indices in L are translated by a simple look-up to $C = \{A[l] \mid l \in L\}$. No value $A[l]$ can occur twice because we would have $R[l'] = l > i$ for the second occurrence l'.

We now turn back to our original selection problem on the generalized suffix

tree $\mathcal{T}$. To achieve an optimal query time, we create an array A containing the document numbers in the order they are encountered at the leaves by a depth first traversal. At the same time, we store the index used for the left-most and for the right-most leaf in the array at each inner node. Assume we have matched a pattern w in the tree and let p be the next node in the tree with $w \in \mathrm{suffixes}(\mathrm{path}(p))$. Let i be the index of the left-most and j be the index of the right-most leaf stored in $\mathcal{T}_p$. Thus, the document numbers of the leaves in the subtree $\mathcal{T}_w$ are those in the subarray $A[i \cdots j]$, and we can retrieve the different document numbers by a CR query on A in linear time in the number of outputs. Chapter 5 contains more applications of range queries.

2.4 Text Indexing Problems

An indexing problem is defined by a database and a query type. We deal here with text indexing problems, so the database is either a single string $t \in \Sigma^*$ or a collection of strings $C \subset \Sigma^*$. A query type defines what the eligible query patterns are and what the output (in dependence on the input pattern) should be. It is possible to define problems where the query is a set of strings, but here we are only interested in the canonical version where the query is a single string called the pattern possibly accompanied by some additional parameters.

In exact pattern matching, the output of a query involves a property on the set of occurrences of a pattern w in the database. We already got to know the text indexing problem in Section 2.3.1, where there is no additional property, i.e., the complete set is to be reported. The solution given there can easily be extended for indexing a set of strings. The problem is then sometimes also called the *occurrence listing problem*. The problem is to preprocess a given collection of texts $S = \{t^1, \ldots, t^n\}$ to answer the following queries: Given a pattern w, report all (i, j) such that ${t^j}_{i,i+|w|-1}$ is equal to w. In Section 2.3.2, we described the document listing problem where the property is the set of document identifiers i of those strings $t^i \in S$ that contain an occurrence of the pattern w.

Besides occurrence and document listing, the third canonical type of problem defined for a collection of strings $S = \{t^1, \ldots, t^n\}$ is the *dictionary indexing problem*. For a query with the pattern w, the desired output is the set of identifiers i such that $t^i = w$. For exact text indexing, optimal algorithms for indexing a single text have been known for a while. The extensions for indexing a set of documents are often straightforward. The use of range queries broadens the possibilities even further (see [Mut02]).

Chapters 4 and 5 deal with the problem of *approximate text indexing*. Approximate text indexing bases its output on the set of approximate occurrences.[1] An *approximate occurrence* of a pattern w in a string t with respect to a maximal distance k in a certain distance model $\mathrm{d} : \Sigma^* \to \Sigma^*$ is a substring $t_{i,j}$ with $\mathrm{d}(t_{i,j}, w) \leq k$. The parameter k

[1] The use of "approximate" here is not to be confused with the use of "approximate" in a lot of other branches of computer science, such as complexity theory. In our context, "approximate" has a well-defined meaning and is a desired property of the solution. On the contrary, for instance, an NP-hard problem is "approximable" if we can find a "solution" that is within a bounded range of the optimum value. Thus, the term "approximate" has a negative touch because it is otherwise practically impossible to compute the "real", optimal solution (unless P=NP).

can either be part of the input or part of the problem instance. The following scheme allows a *classification of the text indexing problems* in this work.

Text Corpus This is either a string $t \in \Sigma^*$ ("Σ^*") or a collection of strings $S \subset \Sigma^*$ ("$\mathcal{P}(\Sigma^*)$").

String Distance The distance is one of the standard distances: edit distance ("edit"), Hamming distance ("hamm"), equality ("exact", for exact indexing), or another distance (e.g., a comparison-based distance).

Distance Bound The bound can be given with the query ("in"), a fixed number ("k"), a proportion with respect to the pattern size ("α"), or another function of the input parameters.

Output Type: We can report occurrences (document number, start and end point; "occ"), positions (document number and start point—corresponding to a suffix of the document starting at the given position; "pos"), or documents (document number only; "doc"). Identifiers of higher hierarchical structures (such as sections, chapters, volumes) are also possible.

Application Rule: We define the suffix ("suff"), the prefix ("pref"), the substring ("substr"), and the entire string rule ("all"). A possible output candidate is reported if the distance between a suffix (prefix, substrings, the entire string) of the candidate and the pattern is below the distance bound.

These five categories allow us to classify each text indexing problem. In the notation of our classification scheme, we denote the document listing problem by the quintuple $\langle \mathcal{P}(\Sigma^*) | \mathsf{exact} | 0 | \mathsf{doc} | \mathsf{substr} \rangle$. The standard edit distance text indexing problem that seeks all positions where an approximate occurrence of the pattern with no more than k errors starts is described by $\langle \Sigma^* | \mathsf{edit} | k | \mathsf{pos} | \mathsf{suff} \rangle$. In Chapter 4, we treat among others the Hamming distance dictionary indexing problem denoted by $\langle \mathcal{P}(\Sigma^*) | \mathsf{hamm} | f(n) | \mathsf{doc} | \mathsf{all} \rangle$ where we seek all strings in S matching a pattern with Hamming distance at most $f(n)$ with $n = |S|$.

2.5 Rice's Integrals

In Chapter 4, we analyze the expected number of nodes visited by a recursive search process. The closed formula that we derive for the expected running time is of the form

$$\sum_{k=m}^{n} \binom{n}{k} (-1)^k f_{n,k} \quad . \tag{2.23}$$

The sum turns out to be of polynomial growth, so we witness an *exponential cancellation effect*. The terms of the sum are exponentially large and alternate in sign. Sums of this type are intimately connected to tries and other digital search trees. Similar problems appear very often in their analysis (see, e.g., [Szp00]). Such sums have also been considered in a general setting by Szpankowski [Szp88a] and Kirschenhofer [Kir96]. It is a common technique to use *Rice's integrals*. The basic idea is captured in the following theorem.

Theorem 2.2 (Rice's integrals).
Let $f(z)$ be an analytic continuation of $f(k) = f_k$ that contains the half line $[m, \infty)$. Then

$$\sum_{k=m}^{n} (-1)^k \binom{n}{k} f_k = \frac{(-1)^n}{2\pi\imath} \oint_{\mathcal{C}} f(z) \frac{n!}{z(z-1)\cdots(z-n)} dz \ , \tag{2.24}$$

where $\mathcal{C}$ is a positively oriented curve that encircles $[m, n]$ and does not include any of the integers $0, 1, \ldots, m-1$ or other singularities of $f(z)$.

The formula already appeared in [Nör24] (Chapter 8, §1), see also [Knu98, FS95, Szp00]. The proof is a simple application of the Cauchy Residue Theorem. We briefly go over some definitions to recapitulate the basic idea. All of this is widely known (see, e.g., [Rud87]). A function is said to be meromorphic in an open set Ω if it is analytic in $\Omega \setminus A$ and has a pole at every point in A, where $A \subset \Omega$ is a set with no limit points in Ω. In essence, a meromorphic function has only countably many poles and is otherwise analytic. For each point $z_0 \in A$ we can "subtract" the pole from the meromorphic function f and get a function

$$f(z) - \sum_{k=1}^{m} \frac{c_k}{(z-z_0)^k} \ , \tag{2.25}$$

which is analytic in the vicinity of z_0. The term $\sum_{k=1}^{m} \frac{c_k}{(z-z_0)^k}$ is called the principle part of f and m is the order of the pole. The complex value c_k is called the *residue* of f at z_1 denoted by

$$\operatorname{res}[f(z), z = z_0] = c_1 \ . \tag{2.26}$$

Let $\mathcal{C}$ be a simple closed curve in Ω encircling the poles $a_1, \ldots, a_n \in A$. The Cauchy Residue Theorem then states that

$$\frac{1}{2\pi\imath} \oint_{\mathcal{C}} f(z) dz = \sum_{k=1}^{n} \operatorname{res}[f(z), z = a_k] \ , \tag{2.27}$$

where $\mathcal{C}$ is traversed in a mathematically positive sense (i.e., counterclockwise).

A brilliant introduction into the technique of Rice's integrals is given in [FS95]. The kernel in (2.24) can also be expressed in terms of Euler's Gamma and Beta functions (see, e.g., [Tem96]) by

$$\begin{aligned} \frac{n!}{z(z-1)\cdots(z-n)} &= \frac{\Gamma(n+1)\Gamma(z-n)}{\Gamma(z+1)} = (-1)^{n+1} \frac{\Gamma(n+1)\Gamma(-z)}{\Gamma(n-z+1)} \\ &= \mathrm{B}(n+1, z-n) = (-1)^{n+1} \mathrm{B}(n+1, -z) \end{aligned} \tag{2.28}$$

If the function $f(z)$ in (2.24) has a moderate (polynomial) growth, the contour-integral can be replaced by a simple integral along a line $\Re(z) = c$ (usually $c = -m + \frac{1}{2}$)

$$\sum_{k=m}^{n} (-1)^k \binom{n}{k} f_k = \frac{1}{2\pi\imath} \int_{c-\imath\infty}^{c+\imath\infty} f(-z) \mathrm{B}(n+1, z) dz \ , \tag{2.29}$$

because, for fixed n, the kernel $\mathrm{B}(n+1, z-n)$ grows asymptotically as $O(z^{-n})$ for $|z| \to \infty$. Details are provided in Section 4.3.2.

To proceed further we either bound the integral or sweep over some residues, very similar to the usage of the Mellin transform. The following *asymptotic expansion* (in the sense of Poincarè) of the ratio of Gamma functions is very helpful and already suggested in [Szp88a].

$$\frac{\Gamma(z+\alpha)}{\Gamma(z+\beta)} \sim \sum_k \frac{\Gamma(\alpha-\beta+1)}{k!\Gamma(\alpha-\beta+1-k)} B_k^{(\alpha-\beta+1)}(\alpha) z^{-k} \quad , \tag{2.30}$$

for all constants α, β with $\beta - \alpha \notin \mathbb{N}$ as $|z| \to \infty$ with $|\arg(z)| < \pi$ [TE51]. The $B_n^{(a)}(x)$ are the *generalized Bernoulli polynomials* (see [Nör24], Chapter 6, §5, or [Tem96]). They are defined by the generating function

$$e^{xt}\left(\frac{t}{e^t-1}\right)^a = \sum_{k=0}^{\infty} B_k^{(a)}(x) \frac{t^k}{k!} \quad , \qquad |t| < 2\pi \quad . \tag{2.31}$$

They are multivariate polynomials in a and x of degree n. The first polynomials are $B_0^{(a)}(x) = 1$, $B_1^{(a)}(x) = -\frac{a}{2} + x$, and $B_2^{(a)}(x) = \frac{3a^2+12x^2-a(1+12x)}{12}$.

For the approximation of $\Gamma(z)\frac{\Gamma(n+1)}{\Gamma(n+1+z)}$ we have to be careful because z is not a constant. Indeed, n has to be regarded as a constant when integrating over z ranging from $c - \imath\infty$ to $c + \imath\infty$. Fortunately, the expansion turns out to be uniform to a certain degree [Fie70]. In this lesser known form, the uniform asymptotic expansion is

$$\begin{aligned}\frac{\Gamma(z+\alpha)}{\Gamma(z+\beta)} &= z^{\alpha-\beta} \sum_{k=0}^{m} \frac{1}{k!} \frac{\Gamma(1+\alpha-\beta)}{\Gamma(1+\alpha-\beta-k)} B_k^{(1+\alpha-\beta)}(\alpha) z^{-k} \\ &\quad + O\left(z^{\alpha-\beta-m}\left(1+|\alpha-\beta|^m\right)\left(1+|\alpha|+|\alpha-\beta|\right)^m\right) \quad ,\end{aligned} \tag{2.32}$$

which is uniformly valid for $|\arg(z+\alpha)| < \pi$, $(1+|\alpha-\beta|)(1+|\alpha|+|\alpha-\beta|) = o(z)$, and $\beta - \alpha \notin \mathbb{N}$ as $z \to \infty$.

Besides the already defined generalized Bernoulli polynomials, we also encounter *generalized Bernoulli numbers* defined by

$$\left(\frac{t}{e^t-1}\right)^a = \sum_{k=0}^{\infty} B_k^{(a)} \frac{t^k}{k!} \quad , \qquad |t| < 2\pi \quad , \tag{2.33}$$

Bernoulli numbers defined by

$$\frac{t}{e^t-1} = \sum_{k=0}^{\infty} B_k \frac{t^k}{k!} \quad , \qquad |t| < 2\pi \quad , \tag{2.34}$$

and *Eulerian polynomials* defined by

$$\frac{1-x}{1-xe^{t(1-x)}} = \sum_{k=0}^{\infty} A_k(x) \frac{t^k}{k!} = \sum_{k=0}^{\infty} \frac{t^k}{k!} \sum_{l=0}^{k} A_{k,l} x^l \quad , \qquad |t| < \frac{\ln x}{x-1} \quad . \tag{2.35}$$

See [Tem96] for the Bernoulli numbers $B_k^{(a)}$ and B_k. See [Com74] for the Eulerian polynomials $A_k(x)$, and [GKP94] for the Eulerian numbers $A_{k,l}$.

Rice's integrals are one of many methods that transfer a discrete problem into the realm of complex analysis. We do not treat (nor use) other methods, but we mention here that another very powerful method is the Mellin transform. Indeed, there is an intimate connection (the Poisson-Mellin-Newton Cycle). We refer to [FR85, FGK$^+$94, FGD95, FS96] for details. For our purposes, Rice's integrals are well suited. In particular, the use of the Mellin transform requires an additional depoissonization step to regain the coefficients [Szp00]. In Chapter 4, we transform our sum to a line integral, which is very similar to the use of the Mellin transform.

Chapter 3

Linear Bidirectional On-Line Construction of Affix Trees

In this chapter, we present a linear-time algorithm to construct affix trees. An affix tree for a string t is a combination of the suffix tree for t and the suffix tree for the reversed string t^R. Besides exhibiting this inherently beautiful duality structure, affix trees allow searching for a pattern alternating the direction of reading, i.e., by starting in the middle and extending to both sides. In contrast, the suffix tree only allows to search a pattern reading from left to right. The alternating searches allow to search for patterns of the kind uvu^R more efficiently using an affix tree. An application is, e.g., the search for certain RNA-related patterns [MP03], e.g., hairpin loops. A hairpin loop is the result of two bonding complementary pieces of an RNA strand. These can be found by searching for patterns of the form uvw, where w is the reversed complementary to u.

We begin this chapter by describing the affix tree data structure and giving the necessary definitions. We continue with a linear-time algorithm to build the affix tree on-line by reading a string only in one direction from left to right (or vice versa). Finally, we describe and analyze the most general case of building the affix tree on-line by appending characters on either side of the string. This case requires a very detailed amortized analysis, which finishes the chapter.

3.1 Definitions and Data Structures for Affix Trees

The affix tree is based on completing the structure exhibited by the suffix links of the suffix tree. Before defining affix trees, we investigate some properties of suffix links that lead naturally to the question of whether and how affix trees can be built.

3.1.1 Basic Properties of Suffix Links

Recall our definition of suffix links from Section 2.3.1. A suffix link points from a node p to a node q if $\mathrm{path}(q)$ is the longest proper suffix of $\mathrm{path}(p)$. For every node p, there can be at most one node representing a longest proper suffix, whereas a node q can represent the shortest proper suffix of multiple other nodes. Because ε is a suffix of every string and is represented by the root, we can add a suffix link for every node.

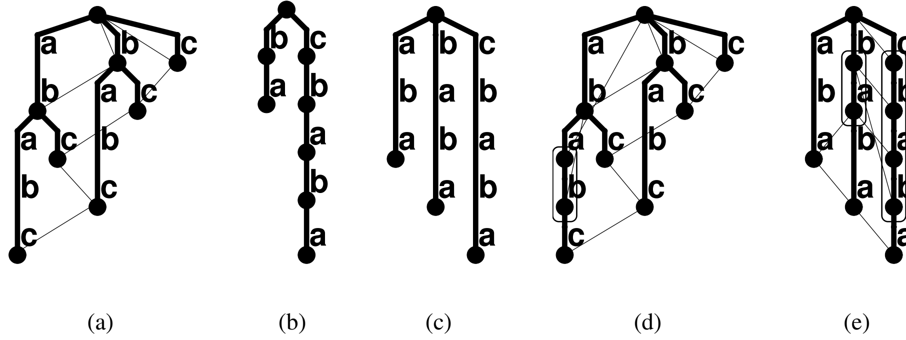


Figure 3.1: Example trees based on the string `ababc`. Figure (a) shows the suffix tree for $t = \texttt{ababc}$, Figure (b) shows the suffix link tree of the suffix tree for $t = \texttt{ababc}$, Figure (c) shows the suffix tree for the reversed string $t = \texttt{cbaba}$, Figure (d) shows a view of the affix tree for $t = \texttt{ababc}$ focused on the suffix structure, and Figure (e) shows the affix tree focused on the prefix structure.

Thus, the suffix links form a tree structure themselves. For a suffix link from p to q, let $\mathrm{path}(p) = u\mathrm{path}(q)$. We label the suffix link with u^R. This way, if we have two suffix links from p to q to r labeled u and v, we get $\mathrm{path}(p) = u^R\mathrm{path}(q)$, $\mathrm{path}(q) = v^R\mathrm{path}(r)$, and $\mathrm{path}(p) = (vu)^R\mathrm{path}(r)$. Hence, we can read and concatenate suffix link labels from the root downward. This leads to the definition of the suffix link tree [GK97].

The *suffix link tree* $\mathcal{T}^R$ of a Σ^+-tree $\mathcal{T}$ that is augmented with suffix links is defined as the tree that is formed by the suffix links of $\mathcal{T}$.

Figure 3.1(a) shows the suffix tree for the string `ababc` with the suffix links from $\overline{\texttt{ab}}$ to $\overline{\texttt{b}}$ to the root as it would have been constructed, e.g., by Ukkonen's algorithm. These links are labeled with `a` and `b`. We added suffix links for the leaves, too, from $\overline{\texttt{ababc}}$ via $\overline{\texttt{babc}}$, $\overline{\texttt{abc}}$, $\overline{\texttt{bc}}$, and $\overline{\texttt{c}}$ to the root. These links are labeled with `a`, `b`, `a`, `b`, and `c`. Figure 3.1(b) shows the resulting suffix link tree.

There is a duality between reversing the string and taking the suffix link tree, proven in [GK97]. For any string t, there is a strong duality for the atomic suffix tree $\mathrm{AST}(t)$ and a weak duality for the (compact) suffix tree $\mathrm{CST}(t)$. In particular, we have $\mathrm{AST}(t)^R = \mathrm{AST}(t^R)$ and $\mathrm{words}(\mathrm{CST}(t)^R) \subseteq \mathrm{words}(\mathrm{CST}(t^R))$.

For the atomic suffix tree, this can easily be seen as follows. Let $t = t_1 \cdots t_n$ and let w be a substring of t. The string w is explicitly represented by a node p in $\mathrm{AST}(t)$ with $w = \mathrm{path}(p) = t_i t_{i+1} t_{i+2} \cdots t_j$. Because $\mathrm{AST}(t)$ is atomic, all edges have length one and all substrings of t have an explicit location. Therefore, there is also a node q with $\mathrm{path}(q) = t_{i+1} t_{i+2} \cdots t_j$ and a suffix link labeled t_i links p to q. By induction, we find a chain of suffix links with labels $t_i, t_{i+1}, t_{i+2}, \ldots, t_j$. Thus, the string $t_j t_{j-1} t_{j-2} \cdots t_i = w^R$ is represented in $\mathrm{AST}(t)^R$. As a result, all substrings of t^R are represented in $\mathrm{AST}(t)^R$. Because $\mathrm{AST}(t^R)$ is defined by $\mathrm{words}(\mathrm{AST}(t^R)) = \mathrm{substrings}(t^R)$, this proves the duality. While this fact about atomic suffix trees is interesting in its own, it only serves as an illustration here. The remaining chapter is only concerned with compact suffix trees. Henceforth, suffix tree

refers only to the compact version again.

For suffix trees, the above construction does not work because not all substrings of t are represented explicitly. The suffix link tree $\mathrm{CST}(t)^R$ represents only a subset of the words of $\mathrm{CST}(t^R)$. In fact, it represents all substrings that have an explicit location in $\mathrm{CST}(t)$.

The following lemmas give an intuition of the part of $\mathrm{substrings}(t)$ that are represented in the suffix link tree $\mathrm{CST}(t)^R$. These results are common knowledge and variations appear throughout the literature. We formalize the results in a suitable way.

Lemma 3.1 (Explicit locations in suffix trees).
In the suffix tree $\mathcal{T}$ for the string t, a substring u is represented by an inner node (its location is explicit) if and only if it is a right-branching substring. It is represented by a leaf if and only if it is a non-nested suffix of t. Finally, the substring is represented by the root if and only if it is the empty string. There are no other nodes in $\mathcal{T}$.

Proof. Clearly, the empty string is always represented by the root.

If $u \in \mathrm{words}(\mathcal{T})$ is a non-nested suffix of t, no $v \in \Sigma^+$ and $w \in \Sigma^*$ exist such that $t = wuv$. If u were not represented by a leaf, there would have to be a node p such that $\mathrm{path}(p) = uv'$ with $v' \in \Sigma^+$. This would be a contradiction to u being non-nested. Therefore, u is represented by a leaf.

If $w \in \mathrm{words}(\mathcal{T})$ is right-branching in t, there exist $a, b \in \Sigma$ with $a \neq b$ and $wa, wb \in \mathrm{substrings}(t)$. The string wa and wb are in $\mathrm{words}(\mathcal{T})$ by the definition of suffix trees. Hence, there are nodes p and q such that wa is a prefix of $\mathrm{path}(p)$ and wb is a prefix of $\mathrm{path}(q)$. From the uniqueness of string locations in Σ^+-trees and from $a \neq b$ it follows that $p \neq q$. Therefore, there must be a branching node r such that r is an ancestor of p and q, and $\mathrm{path}(r)$ is a prefix of $\mathrm{path}(p)$ and $\mathrm{path}(q)$. Therefore, $\mathrm{path}(r)$ must be a prefix of w. If $\mathrm{path}(r) \neq w$, there would be two edges leading out of r with labels starting with the same character, which would contradict $\mathcal{T}$ being a Σ^+-tree. As a result, $\mathrm{path}(r) = w$ and w is represented by a node.

Let p be a node in $\mathcal{T}$. If p is not the root (representing the empty string), by the compactness of $\mathcal{T}$, it must be either a leaf or a branching node.

If p is a leaf, it represents a substring $u = \mathrm{path}(p)$ of t. Suppose u is not a suffix, then the string uw with $w \in \Sigma^+$ is in $\mathrm{substrings}(t)$ and consequently in $\mathrm{words}(\mathcal{T})$. There must be a node q with path $\mathrm{path}(q) = uw$, and hence p cannot be a leaf because it must be the parent of q. Suppose u is a nested suffix. By definition, we have $u \in \mathrm{psubstrings}(t) \cup \mathrm{prefixes}(t)$. Again there must be $w \in \Sigma^+$ with $uw \in \mathrm{substrings}(t)$—leading to the same contradiction. As a result, $\mathrm{path}(p)$ must represent a non-nested suffix.

If p is a branching inner node, there must be two nodes q and r which are children of p. Let $w = \mathrm{path}(p)$. Because all outgoing edges of p have labels with different start characters, we have $\mathrm{path}(q) = wau$ and $\mathrm{path}(r) = wbv$ for $u, v \in \Sigma^*$. Therefore, wa, wb are substrings of t and w is right-branching in t. □

A very important consequence of the previous lemma and Observation 2.1 is the following.

Lemma 3.2 (Chain property of suffix links).
Let p be a node in the suffix tree $\mathcal{T}$ for the string t. The node p is either the root, the leaf representing the shortest non-nested suffix, or p has an atomic suffix link.

Proof. Let p be a node that is not the root and not the node representing the shortest non-nested suffix of t. Let $u = \mathrm{path}(p)$. If p is a leaf, it represents a non-nested suffix. Because p does not represent the shortest non-nested suffix, there is a shorter non-nested suffix v such that $av = u$ for some $a \in \Sigma$. Since v is a non-nested suffix, there is a node q representing v, and there is an atomic suffix link from p to q.

If p is not a leaf (and not the root), it must be a branching node. Therefore, u is right-branching in t. Let $v \in \Sigma^*$ be a proper suffix of u such that $av = u$ for some $a \in \Sigma$. By Observation 2.1, v is also right-branching and, by Lemma 3.1, represented by a node q. Therefore, there is an atomic suffix link from p to q. □

From the above lemmas, we know that the suffix link tree $\mathcal{T}^R$ of a suffix tree $\mathcal{T}$ for the string t represents a subset of all substrings of t^R, namely the left-branching substrings and t^R itself (see Figures 3.1(b) and 3.1(c)). The idea of an affix tree is to augment a suffix tree with nodes such that it represents all suffixes of t and that its reverse represents all suffixes of t^R (respectively the prefixes of t).

3.1.2 Affix Trees

Affix trees can be defined as "dual" suffix trees. Formally, an *affix tree* $\mathcal{T}$ for a string t is a Σ^+-tree with suffix links such that

$$\begin{aligned} \mathrm{words}(\mathcal{T}) &= \mathrm{substrings}(t) \qquad \text{and} \\ \mathrm{words}\left(\mathcal{T}^R\right) &= \mathrm{substrings}\left(t^R\right) . \end{aligned}$$

Figures 3.1(d) and 3.1(e) show the affix tree for `ababc`. Figure 3.1(d) focuses on the suffix-tree-like structure and Figure 3.1(e) on the prefix-tree-like structure. The differences to the suffix tree and to the prefix tree in each structure are the additional nodes that are boxed (compare (a), (c), (d), and (e) in Figure 3.1).

Note that a substring w of t might have two different locations in the affix tree $\mathcal{T}$ for t: the location of w in the suffix tree part and the location of w^R in the prefix tree part. If these locations are implicit, then they are not represented by the same node but lie "in" two different edges. For example, the location of `bab` appears in the suffix edge between the nodes $\overline{\mathtt{b}}$ and $\overline{\mathtt{babc}}$ (Figure 3.1(d)) and in the prefix edge between the nodes $\overline{\mathtt{ba}}$ and $\overline{\mathtt{baba}}$ (Figure 3.1(e)). The location of `ba` appears in the suffix edge between the nodes $\overline{\mathtt{b}}$ and $\overline{\mathtt{babc}}$ (Figure 3.1(d)) and in the prefix edge between the root and $\overline{\mathtt{aba}}$ (Figure 3.1(e)).

Having two different implicit locations complicates the process of making a location explicit (i.e., inserting a node for the location). Both implicit locations must be found for that. To insert a node $\overline{\mathtt{ba}}$ (respectively $\overline{\mathtt{ab}}$ in the prefix view) into the affix tree shown in Figures 3.1(d) and 3.1(e), the two edges $(\overline{\mathtt{b}}, \overline{\mathtt{babc}})$ and $(\mathrm{root}, \overline{\mathtt{aba}})$ have to be split.

The affix tree for a string t represents two trees in one. It represents the suffix tree for t and the suffix tree for t^R. The latter is called the *prefix tree* for the string t. From Lemma 3.1, we learned that the affix tree contains all nodes that would be part of the suffix tree and all nodes that would be part of the prefix tree. As an application of this lemma, one can classify each node whether it belongs to the suffix part, the prefix part, or both parts. Thus, we define the following node type. A node is a *suffix node* if it is a suffix leaf (it has no outgoing suffix edges), if it is the root, or if it is a

branching node in the suffix tree (it has at least two outgoing suffix edges). A node is a *prefix node* if it is a prefix leaf (it has no outgoing prefix edges), if it is the root, or if it is a branching node in the prefix tree (it has at least two outgoing prefix edges). The attribute needs not be stored explicitly but can be reconstructed on-line. Edges are either suffix or prefix edges. The suffix links are the (reversed) prefix edges and vice versa. Therefore, all edges need to be traversable in both directions. In the following, we distinguish between the suffix structure and the prefix structure of the affix tree.

Throughout the remaining chapter, let $\mathrm{ST}(t)$ denote the suffix tree and let $\mathrm{AT}(t)$ denote the affix tree built from the string t, which we call the *underlying string*. The prefix tree can be denoted by $\mathrm{ST}(t^R)$.

3.1.3 Additional Data for the Construction of Affix Trees

An essential property of affix trees is its inherent duality. In the final sections of this chapter, we show how to build the affix trees by expanding a string in both directions. To support such a behavior, we can implement a string as a pair of dynamic tables (see Section 2.2.2). The first table contains the positive indices (starting at zero) and the second table the negative indices (starting at -1). When switching the view (from suffix to prefix), a simple translation $f : x \to -1 - x$ also switches the indices and allows to read the string in a completely reversed manner. Thus, on the implementation level we number strings differently, starting at zero. To distinguish this case, we denote the characters of a string t by $t[i]$, where i ranges from zero to $n - 1$.

For the algorithmic part, we need to be able to keep track of locations in the suffix or affix tree and be able to reach these quickly. In Section 2.1.2, we introduced virtual nodes as means for referencing implicit locations. A reference pair is a representation of a virtual node of a suffix tree introduced in [Ukk95] for that reason. For a suffix tree $\mathcal{T}$ for the string t, a *reference pair* is a pair $(p, (o, l))$ where p is a node (called the *base* of the reference pair), and (o, l) represents a substring $u = t_{o,o+l-1}$. It corresponds to the virtual node (p, u). For ease of notation, we also denote the reference pair $(p, (o, l))$ by (p, u) knowing that u is a substring of t and can be represented by (o, l). To actually reach the location, we start at the base and move along edges whose labels represent the same string as u.

As mentioned above, there may be two locations of a substring of t in the affix tree, one in the suffix structure and one in the prefix structure. We therefore need to distinguish between a suffix reference pair, where the represented location is reached by moving along suffix edges, and a prefix reference pair, where the represented location is reached by moving along prefix edges. Conceptually, one can also think of using a combined reference pair for a string in the affix tree made up of a prefix and a suffix reference pair.

For the construction of suffix trees, a reference pair (p, u) is called *canonical* if there exists no node q and a suffix $v \in \mathrm{suffixes}(u)$ such that $\mathrm{path}(p)$ is a proper prefix of $\mathrm{path}(q)$ and $\mathrm{path}(p)u = \mathrm{path}(q)v$. For the construction of affix trees, we use suffix and prefix reference pairs. For a suffix reference pair (p, u), we require q to be a suffix node, and, for a prefix reference pair (p, u), we require p to be a prefix node. A suffix reference pair is canonical if there is no suffix node q and a string v such that $\mathrm{path}(p)$ is a proper prefix of $\mathrm{path}(q)$ and $\mathrm{path}(p)u = \mathrm{path}(q)v$. In other words, a canonical reference pair is a reference pair where the base node has maximal depth.

For example, in Figure 3.1 with the underlying string $t = t[0] \cdots t[4] = \texttt{ababc}$ (with $t^R = t[-1] \cdots t[-5] = \texttt{cbaba}$) the canonical prefix reference pair for ab is $(\mathrm{root}, (-3, 2))$, the canonical suffix reference pair for $\texttt{ba} = (\texttt{ab})^{-1}$ is $(\overline{\texttt{ab}}, (3, 0))$. A non-canonical suffix reference pair for ba is $(\mathrm{root}, (1, 2))$.

Similar to suffix trees, edges are implemented with indices into the underlying string t. For the efficient handling of leaves, we also use *open edges* as introduced in [Ukk95]. The edges leading to leaves are labeled with the starting index and an infinite end index (e.g., (i, ∞)) that represents the end of the underlying string. Such an edge always extends to the end of the string. This can easily be achieved using a single bit (or by implicitly assuming the end index ∞ for leaves).

For the on-line construction of affix trees, we need to keep track of some important locations in the tree. Let $\alpha(t)$ be the *longest nested suffix* of the string t and $\hat{\alpha}(t)$ the *longest nested prefix* of t. Obviously, we have $\alpha(t) = (\hat{\alpha}(t^R))^R$. The longest nested suffix is called the *active suffix* and the longest nested prefix is called the *active prefix*. By Lemma 3.1, all longer suffixes (respectively prefixes) are represented by leaves. As with suffix trees, all updating is done "around" the positions of the active suffix and the active prefix. We therefore always keep the following data for the construction of compact affix trees:

- A canonical suffix reference pair $asp_{\mathsf{suffix}}(t)$ for the active suffix point, the suffix location of the active suffix.
- A canonical prefix reference pair $asp_{\mathsf{prefix}}(t)$ for the active suffix point, the prefix location of the active suffix, which is the active prefix of the reverse string.
- A canonical prefix reference pair $app_{\mathsf{prefix}}(t)$ for the active prefix point, the prefix location of the active prefix. The active prefix is the active suffix of the reverse string, so it is the counterpart to $asp_{\mathsf{suffix}}(t)$ in the prefix view.
- A canonical suffix reference pair $app_{\mathsf{suffix}}(t)$ for the active prefix point, the suffix location of the active prefix.
- The active suffix leaf $asl_{\mathsf{suffix}}(t)$, which represents the shortest non-nested suffix of t.
- The active prefix leaf $apl_{\mathsf{prefix}}(t)$, which represents the shortest non-nested prefix of t.

For the four reference pairs and the two leaves we also keep the "depth" (the length of the represented string). This is needed in order to calculate the correct indices of edge labels in the implementation of the algorithm.

Note that $asp_{\mathsf{prefix}}(t)$ should be read as "the location of the active suffix in the prefix view". In this way, we fix a view and the definition of active suffix and active prefix refers to the string t and not to its reverse. When reversing the view, suffix and prefix also have to be reversed, e.g., $app_{\mathsf{suffix}}(t)$ becomes $asp_{\mathsf{prefix}}(t^R)$.

As already mentioned before, we can conceptually see $asp_{\mathsf{suffix}}(t)$ and $asp_{\mathsf{prefix}}(t)$ as one combined reference pair for the active suffix and $app_{\mathsf{prefix}}(t)$ and $app_{\mathsf{suffix}}(t)$ as one combined reference pair for the active prefix.

A non-nested suffix, represented by a suffix leaf in the suffix tree, is not represented by a node in the prefix tree because it occurs only once as the prefix of t^R. Hence, all suffix leaves must be hidden in a prefix tree-like view of the affix tree.

By Lemma 3.2, the asl_{suffix} is the only suffix node other than the root that may have a non-atomic suffix link. Therefore, all suffix leaves form a chain with atomic suffix links because, if a leaf is not the asl_{suffix}, there is a leaf representing a suffix that is shorter by one character. This chain of suffix leaves must be hidden in the prefix view. Fortunately, the chain does not extend over the leaves to other suffix nodes. The next lemma proves that the prefix parent of asl_{suffix} is always a prefix node, too, and is hence represented by a node in the prefix tree.

Theorem 3.3 (Parent of the active suffix leaf).
The prefix (or suffix) parent of the active suffix (respectively prefix) leaf is always a prefix (respectively suffix) node.

Proof. The $asl_{\mathsf{suffix}}(t)$ is the leaf representing the shortest non-nested suffix. Let p be the prefix parent of $asl_{\mathsf{suffix}}(t)$.

Suppose p is not a prefix node, then p must be a suffix node and $w = \mathrm{path}(p)$ is right-branching in t. By the definition of suffix links, w is a suffix of $\mathrm{path}(asl_{\mathsf{suffix}}(t))$ and hence a suffix of t. For w to be right-branching there must be at least two occurrences of w in t, one followed by $a \in \Sigma$, the other by $b \in \Sigma$ (with $a \neq b$). Taking a look at the preceding characters, we therefore find that

$$xw \in \mathrm{suffixes}\,(t) \quad \text{and} \quad ywa, zwb \in \mathrm{substrings}\,(t) \tag{3.1}$$

with $x, y, z \in \Sigma \cup \{\epsilon\}$ and not both y and z are empty strings (otherwise $a = b$). Note that there can be no node q with $u = \mathrm{path}(q)$ such that u is a proper suffix of $\mathrm{path}(asl_{\mathsf{suffix}}(t))$ and w is a proper suffix of u. We say that such a node q "lies between" p and $asl_{\mathsf{suffix}}(t)$. Because $asl_{\mathsf{suffix}}(t)$ is represented by a leaf and p is its direct suffix parent, this node would be between these two, contradicting the original assumptions.

If y and z are different characters, then w is left-branching and—by Lemma 3.1—a prefix node. Suppose the contrary is true, then we can distinguish two cases: Either y and z are equal, or one of the occurrences of w is as a prefix of t, i.e., $y = \varepsilon$ or $z = \varepsilon$.

Let us *first* assume that both are equal and let $y = z = c \in \Sigma$. Either x is equal to c or it is different. If we had $x = c$, then cw would be right-branching and represented by a suffix node q lying between p and $asl_{\mathsf{suffix}}(t)$—a contradiction to our assumptions. If we have $x \neq c$ then w would be left-branching and p is a prefix node.

Thus, we find ourselves in the *second* case: One of y, z must be the empty string. Hence, w occurs as a prefix of t. Without loss of generality, assume that $z = \varepsilon$. We then have the situation

$$xw \in \mathrm{suffixes}\,(t),\ ywa \in \mathrm{substrings}\,(t), \text{ and } wb \in \mathrm{prefixes}\,(t)\ . \tag{3.2}$$

If $x \neq y$, then w is left-branching and p is a prefix node. Therefore, we must have $x = y = c$ for $c \in \Sigma^+$. Taking a look at the next position to the left of c we find that

$$xcw \in \mathrm{suffixes}\,(t),\ ycwa \in \mathrm{substrings}\,(t), \text{ and } wb \in \mathrm{prefixes}\,(t) \tag{3.3}$$

for yet different $x, y \in \Sigma \cup \{\epsilon\}$.

Again we find that, if $x \neq y$, we have a left-branching string u that is represented by a prefix node lying between p and $asl_{\mathsf{suffix}}(t)$. Thus, either $x = y$ or y is the empty string (since $ycwa$ is closer to the left string border than xcw). Iterating this argument leads to

$$cuw \in \text{suffixes}\,(t) \quad \text{and} \quad uwa, wb \in \text{prefixes}\,(t) \ , \tag{3.4}$$

for some string $u \in \Sigma^+$ (at some point the left border is reached, y is the empty string, and x is the character c). Therefore, wb is a prefix of uw. (Clearly, we have $|u| > 0$ because otherwise $a = b$.) It follows that cwb is a substring of t. Now, let $u = vd$ for some character $d \in \Sigma$, thus $vdwa$ is a prefix of t, and the situation is

$$cwb, dwa \in \text{substrings}\,(t) \quad \text{and} \quad dw \in \text{suffixes}\,(t) \ . \tag{3.5}$$

If $c = d$, then dw is a right-branching string represented by a node q that lies between p and $asl_{\mathsf{suffix}}(t)$—a contradiction. Therefore, $c \neq d$, w is left-branching, and p is a prefix node. □

From Lemma 3.2 and the above theorem, we conclude that all nodes that occur in the prefix tree for t but not in the suffix tree for t (so called *prefix only nodes*) appear in chains with atomic edges limited by suffix nodes. As a consequence, we can cluster the prefix nodes in the affix tree that would not be part of the suffix tree. Let p be a prefix node that is not a suffix node. The node p has exactly one suffix child and one suffix parent. If the suffix parent or the suffix child of p is also a prefix only node, the edge length must be one. For the suffix structure, the prefix nodes can thus be clustered to *paths*, where the nodes are connected by edges of length one. It suffices to know the number of nodes in the path to know the string length of the path. What is an edge in the suffix tree becomes an edge leading to a non-suffix node, a path of prefix nodes, and a final edge leading again to a suffix node in the affix tree. See Figures 3.1(d) and 3.1(e) where the nodes are already in boxes corresponding to their paths.

The path structure has the following properties:

- All non-suffix (or non-prefix) nodes in the suffix (or prefix) structure are clustered in paths.
- Accessing the i-th (in particular the first and the last) node takes constant time.
- The first and the last node of each path hold a reference to the path and know the path length.
- A path can be split into two in constant time similar to an edge. (This is necessary, e.g., when a prefix only node in a path becomes a suffix node.)
- Nodes can be dynamically appended and prepended to a path.

With the first three properties, the affix tree can be traversed in the same asymptotic time as the corresponding suffix tree and, with the latter two properties, the updating also works in the same asymptotic time.

The affix tree can be traversed ignoring nodes of the reverse kind: If a non-suffix node is hit while traversing the suffix part, it must have a pointer to a path. The last node of that path is again a prefix node with a single suffix node as suffix child. This is the key to achieving a linear time bound in the construction of affix trees.

3.1.4 Implementation Issues

We briefly take a closer look at the details of the data structures. The edges need to be traversable in both directions. Therefore, we store a starting and ending point and the indices (including a flag to mark open edges) in the underlying string at each edge. To traverse the edges in both directions, we need to store the outgoing suffix and prefix edges indexed by their starting character and the suffix and prefix parent edges at the nodes. The first and the last node of a path need to have a reference to the path. A node cannot be part of two paths, so each node carries a single path pointer. The type of a path can be derived from the node type, which in turn can be derived from the number of children of each type. The adjacency list can be stored in various ways (trees, lists, arrays, hash tables). The optimal storage depends on the application. For the analysis, the alphabet is assumed to be fixed and the only thing needed is a procedure $\mathrm{getEdge}(\textit{node}, \textit{type}, \textit{starting-character})$ that returns the edge of the given type and with the given character.

The paths are stored as one or two dynamic arrays that can grow with constant amortized cost per operation (see Section 2.2.2). The size of the path is linear in the number of nodes it contains. Paths never need to be merged: Except for the active prefix leaf, no nodes are ever deleted. Between the active prefix leaf and the next prefix node closer to the root, there is always one suffix node (see Theorem 3.3), so the two clusters can never merge. An inner node that is of suffix type represents a right-branching substring w of t. Regardless of appending or prepending to t, the substring w always stays branching.

To be able to split paths, we use path pointers. A path pointer has a reference to the path, an index to the first node pointer, and a length. When a path is split, two path pointers share the same dynamic array structure. For example, the nodes $\overline{\texttt{ab}}$, $\overline{\texttt{aba}}$, $\overline{\texttt{abab}}$, $\overline{\texttt{ababa}}$ are in a common path in the affix tree for $t = \texttt{aabababa}$ (Figure 3.2(a)). After the affix tree is extended to $t = \texttt{aabababaa}$ (Figure 3.2(b)), the nodes $\overline{\texttt{aba}}$ and $\overline{\texttt{ababa}}$ are suffix nodes and no longer path members. The nodes $\overline{\texttt{ab}}$ and $\overline{\texttt{abab}}$ now share the same dynamic array, but the array does not need to grow because the nodes $\overline{\texttt{a}}$, $\overline{\texttt{aba}}$, and $\overline{\texttt{ababa}}$ will never be deleted, and no other nodes can be inserted between them.

The size of an affix tree depends on the way the children of a node are stored. Assuming a constant size alphabet Σ, it makes no difference to the asymptotic running time whether we use linear lists, hash tables, or arrays. For a small alphabet Σ (e.g., $|\Sigma| = 5$) an array is probably the smallest and fastest solution. In that case, the size of an affix tree for a string of size n can be estimated as follows: Add the size of the two parent edges to the node to that they lead, so for each node we count the two parent edges (each with two indices, one open edge flag, one type flag, and two pointers), the two children arrays (each $|\Sigma|$ pointers), the two parent pointers (one pointer each), a path pointer (two indices, one pointer), and a path array structure (two pointers to dynamic arrays, five indices for size, length and a reference count, and one pointer to a node). The number of nodes is at most $2n$. The size of the underlying string (as a two way extensible dynamic array structure) is at most $2n$ characters. The size of the reference pairs does not contribute asymptotically. An upper bound for the total size of the affix tree $\mathcal{T}$ is then $\mathrm{size}(\mathcal{T}) \leq n(22 \text{ indices} + 2 \text{ characters} + 8 \text{ flags} + (22 + 4|\Sigma|) \text{ pointers})$. Assuming a four byte machine architecture (storing flags in a bit of

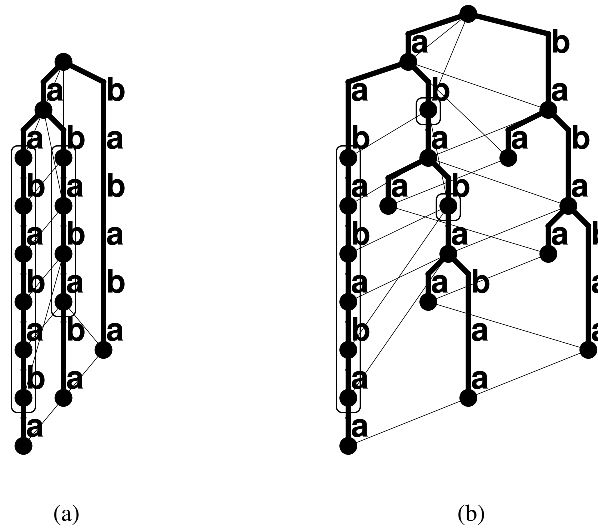


Figure 3.2: The affix trees for `aabababa` and for `aababababaa`.

some index or pointer, characters in one byte, and pointers and indices in four bytes) and a four element alphabet, the size is then at most $250n$ bytes.

3.2 Construction of Compact Suffix Trees

In order to prepare the construction of affix trees, we briefly describe how to construct suffix trees by expanding the underlying string once to the right and once to the left.

These problems are solved by two already existing algorithms, namely Ukkonen's algorithm [Ukk95] and Weiner's algorithm [Wei73]. We assume some additional information in the latter algorithm readily available when constructing affix trees; therefore, the described algorithm is not identical to Weiner's algorithm (but simpler). The results of this section are due to Ukkonen and Weiner. We replicate them in detail here to prepare the presentation of the affix tree construction algorithm.

3.2.1 On-Line Construction of Suffix Trees

Ukkonen's algorithm constructs a suffix tree on-line from left to right, building $\mathrm{ST}(ta)$ from $\mathrm{ST}(t)$. The algorithm can intuitively be derived from an algorithm that constructs atomic suffix trees. It essentially works by lengthening all suffixes represented in $\mathrm{ST}(t)$ by the character a and inserting the empty suffix. Obviously, the resulting tree is $\mathrm{ST}(ta)$.

To analyze the algorithm the suffixes are divided into three different types:

Type 1. Non-nested suffixes of t.

Type 2. Nested suffixes s of t, where sa is a substring of t.

Type 3. Nested suffixes s of t, where sa is not a substring of t. We call sa a *relevant suffix*.

To extend t by the character a, we deal with each of these sets of suffixes:

Case 1. For all suffixes s of t of Type 1, sa is non-nested in ta (otherwise sa would occur elsewhere in t and s were nested). The suffix s was represented by a leaf (Lemma 3.1) in $\mathrm{ST}(t)$ and sa will be represented by a leaf in $\mathrm{ST}(ta)$. With the use of open edges for leaves, all leaves will extend automatically to the end of the string. Nothing needs to be done for these suffixes.

Case 2. For all suffixes s of t of Type 2, sa is a substring of t. Hence, sa is a nested suffix of ta. Nothing needs to be done for these suffixes.

Case 3. For all suffixes s of t of Type 3, sa is a non-nested suffix and must thus be represented by a leaf in $\mathrm{ST}(ta)$. The suffix s is not represented by a leaf in $\mathrm{ST}(t)$.

If sa is a relevant suffix (Type 3), then s is a nested suffix of t. Therefore, s must be a proper suffix of t and s occurs as a substring elsewhere in t. The extended suffix sa is non-nested in ta, so sa cannot occur in t. Therefore, there is a substring sb in t with $b \neq a$. Thus, $sa, sb \in \mathrm{substrings}(t)$ and s is right-branching in ta. To update $\mathrm{ST}(t)$ with the relevant suffix sa, a node $\overline{s}$ must be inserted (unless s was right-branching in t and $\overline{s}$ already exists) and a new leaf $\overline{sa}$ must be added at the (possibly new) node $\overline{s}$.

Lemma 3.4.
Every relevant suffix sa is a suffix of $\alpha(t)a$, and $\alpha(ta)$ is a suffix of every relevant suffix.

Proof. The active suffix $\alpha(t)$ is the longest nested suffix of t. If sa is a relevant suffix, s is a nested suffix of t. Hence, s is a suffix of $\alpha(t)$ and sa a suffix of $\alpha(t)a$. The new active suffix $\alpha(ta)$ cannot be longer than $\alpha(t)a$. Suppose otherwise that it would be longer, then $\alpha(ta) = v\alpha(t)a$ for some $v \in \Sigma^+$. Because the new active suffix $\alpha(ta)$ is nested in the new string ta, we have $v\alpha(t)a \in \mathrm{psubstrings}(ta) \cup \mathrm{prefixes}(ta)$. It follows that $v\alpha(t) \in \mathrm{psubstrings}(t) \cup \mathrm{prefixes}(t)$ and $v\alpha(t) \in \mathrm{suffixes}(t)$. The suffix $v\alpha(t)$ would be nested and longer than $\alpha(t)$. This contradicts the definition of the active suffix.

Both, $\alpha(ta)$ and $\alpha(t)a$ are suffixes of ta, therefore $\alpha(ta)$ is a suffix of $\alpha(t)a$. Because a relevant suffix sa is non-nested in ta, it is longer than $\alpha(ta)$. Hence, $\alpha(ta)$ is a suffix of sa. □

Corollary 3.5.
The new active suffix $\alpha(ta)$ is a suffix of $\alpha(t)a$.

Let $\alpha(t) = t_{i_n,n}$ and let $\alpha(ta) = t_{i_{n+1},n}a$. Let us denote the relevant suffixes by $s_k = t_{i_n+k,n}$ with $k \in \{0, \ldots, i_{n+1} - i_n - 1\}$ (if $i_n = i_{n+1}$, there is no relevant suffix). The (canonical) reference pair $asp_{\mathsf{suffix}}(t) = (p, (o, l))$ represents the active suffix $\alpha(t)$ such that $\alpha(t) = \mathrm{path}(p)t_{o,o+l-1}$ (here, $o + l - 1 = n$). Therefore, p represents the prefix $t_{i_n,o-1}$ of $\alpha(t)$. For all relevant suffixes, we can reach the location of s_k with $asp_{\mathsf{suffix}}(t)$ as follows. The first relevant suffix is immediately available because we

Procedure 3.3 canonize(*reference pair* $(p,(o,l))$)

Parameters: $(p,(o,l))$ is a reference pair that is to be canonized.

```
1: if l ≠ 0 then {Otherwise the reference pair is already canonical.}
2:    e := getEdge(p, SUFFIX, t_o);
3:    k := getLength(e);
4:    while k ≤ l do
5:       o := o + k;
6:       l := l − k;
7:       p := getTargetNode(e);
8:       e := getEdge(p, SUFFIX, t_o);
9:       k := getLength(e);
```

have $s_0 = \mathrm{path}(p)t_{o,o+l-1}$ (with $n = o + l - 1$). We reach the other locations by modifying the reference pair. Let $(p_k,(o_k,l_k))$ be the canonical reference pair[1] for the location of the relevant suffix s_k. To reach the location of s_{k+1} from the reference pair $(p_k,(o_k,l_k))$, we have two cases. If p_k is not the root, it has a suffix link to a node q such that $\mathrm{path}(p_k) = t_{i_n+k}\mathrm{path}(q)$ and $s_{k+1} = \mathrm{path}(q)t_{o_k,o_k+l_k-1} = t_{i_n+k+1,n}$. The new base is canonized (see Procedure 3.3), which changes the base to p_{k+1} and the offset and length to o_{k+1} and l_{k+1}. Otherwise, if p_k is the root, then $s_k = t_{o_k,o_k+l_k-1}$, and we can get to the location of s_{k+1} by increasing the offset o and decreasing the length l because we have $s_{k+1} = t_{o_k+1,(o_k+1)+(l_k-1)-1}$. Again we invoke the procedure canonize(). The relevant suffixes are thus reached by starting at $asp_{\mathsf{suffix}}(t)$ and iteratively moving over suffix links and canonizing. It can easily be determined whether we have reached a non-relevant suffix (Type 2) by checking whether the current reference pair asp_{suffix} can be extended by a. If this is the case, the current represented suffix is nested, and the current reference pair is $asp_{\mathsf{suffix}}(ta)$.

The suffix links for the new nodes can be set during the process or afterwards if we store each node p_k inserted (or found) during the handling of s_k on a stack. Node p_k from the k-th iteration of the while loop is linked to the node p_{k+1} of the following iteration. The last node $p_{i_{n+1}-i_n-1}$ gets a suffix link to an already existing node (called the *end point*), which represents a string one character shorter than $\alpha(ta)$ and is either the base of $asl_{\mathsf{suffix}}(ta)$ or one edge below the base. By Lemma 3.2, the end point always exists.

Procedure 3.3 takes a reference pair $(p,(o,l))$ and changes the base, the offset, and the length so that the resulting reference pair represents the same string and is canonical. The process is called *canonizing* a reference pair. This is done by following edges that represent suffixes of the string part of the reference pair down the tree until the node is closest to the represented location.

Lemma 3.6.
The amount of work performed during all calls to canonize() *with the reference pair* $asp_{\mathsf{suffix}}(t)$ *is* $O(|t|)$.

[1] The reference pairs $(p_k,(o_k,l_k))$ do not exist on their own but are stages reached by the reference pair $asp_{\mathsf{suffix}}(t)$ before it becomes $asp_{\mathsf{suffix}}(ta)$.

Proof. Let $n = |t|$. The amount of work performed during canonizing the reference pair $asp_{\mathsf{suffix}}(t)$ can be bounded as follows. Each time a character is appended to t, a coin is put on the character. The reference pair $asp_{\mathsf{suffix}}(t)$ consists of a base p, a start index o into t, and a length l. The string part of the reference pair is $t_{o,n}$ leading to the implicit location. Each time the base is moved along an edge e down the tree, the length l is decreased and the index o is increased by the length k of the edge. We take the coin from character t_o to pay for the step. Because o is never decreased, there is always a coin on t_o. As a result, canonizing takes amortized constant time per call. □

Theorem 3.7.
Ukkonen's algorithm constructs $\mathrm{ST}(t)$ *on-line in time* $O(|t|)$.

Proof. Let $n = |t|$. The only part with non-constant running time per iteration is the insertion of the relevant suffixes. The number of nodes of $\mathrm{ST}(t)$ is at most $2n$, the number of leaves is at most n (there are at most n leaves, so there cannot be more than n branching nodes). Each time (but once per iteration) that a suffix link is taken, at least one leaf is inserted. The number of links taken is therefore less than n. The amount of work performed during canonizing the reference pair $asp_{\mathsf{suffix}}(t)$ is $O(n)$ by Lemma 3.6. □

See Figure 3.4 which shows the intermediate steps of the on-line construction of $\mathrm{ST}(\mathtt{acabaabac})$ from $\mathrm{ST}(\mathtt{acabaaba})$. The reference pair starts out with the base $\bar{\mathtt{a}}$ and the remaining string ba, e.g., as $(\bar{\mathtt{a}}, (6, 2))$ (the indices are implementation indices starting at zero in this example). After the insertion of the new leaf between (b) and (c), the suffix link from $\bar{\mathtt{a}}$ to the root is taken. The edge starting with b has length six, and thus no canonizing takes place. After the insertion of the second leaf between (c) and (d), the offset is increased, the length decreased (instead of taking a suffix link). The resulting reference pair is $(\mathrm{root}, (5, 1))$ with remaining string a. The reference pair is then canonized to $(\mathtt{a}, (6, 0))$, where it can be extended to $(\mathtt{a}, (6, 1))$ with the remaining string c.

3.2.2 Anti-On-Line Suffix Tree Construction with Additional Information

Anti-on-line construction builds $\mathrm{ST}(at)$ from $\mathrm{ST}(t)$. The tree $\mathrm{ST}(t)$ already represents all suffixes of $\mathrm{ST}(at)$ except at. Only the suffix at needs to be inserted. The leaf for at will branch at the location representing $\hat{\alpha}(at)$ in $\mathrm{ST}(t)$.

We assume that we have the additional information of knowing the length of $\hat{\alpha}(at)$ available in each iteration of the following algorithm. With this, we do not need the "indicator vector" used by Weiner [Wei73] to find the length of the new active prefix.

There are three possibilities. The node location may be represented by an inner node, the location is implicit in some edge, or the location is represented by a leaf. In the last case, the new leaf does not add a new branch to the tree but extends an old leaf that now represents a nested suffix. The node representing the old leaf needs to be deleted after attaching the new leaf to it (it represents a nested suffix now, see Lemma 3.1).

Lemma 3.8.
The new active prefix $\hat{\alpha}(at)$ *is a prefix of* $a\hat{\alpha}(t)$.

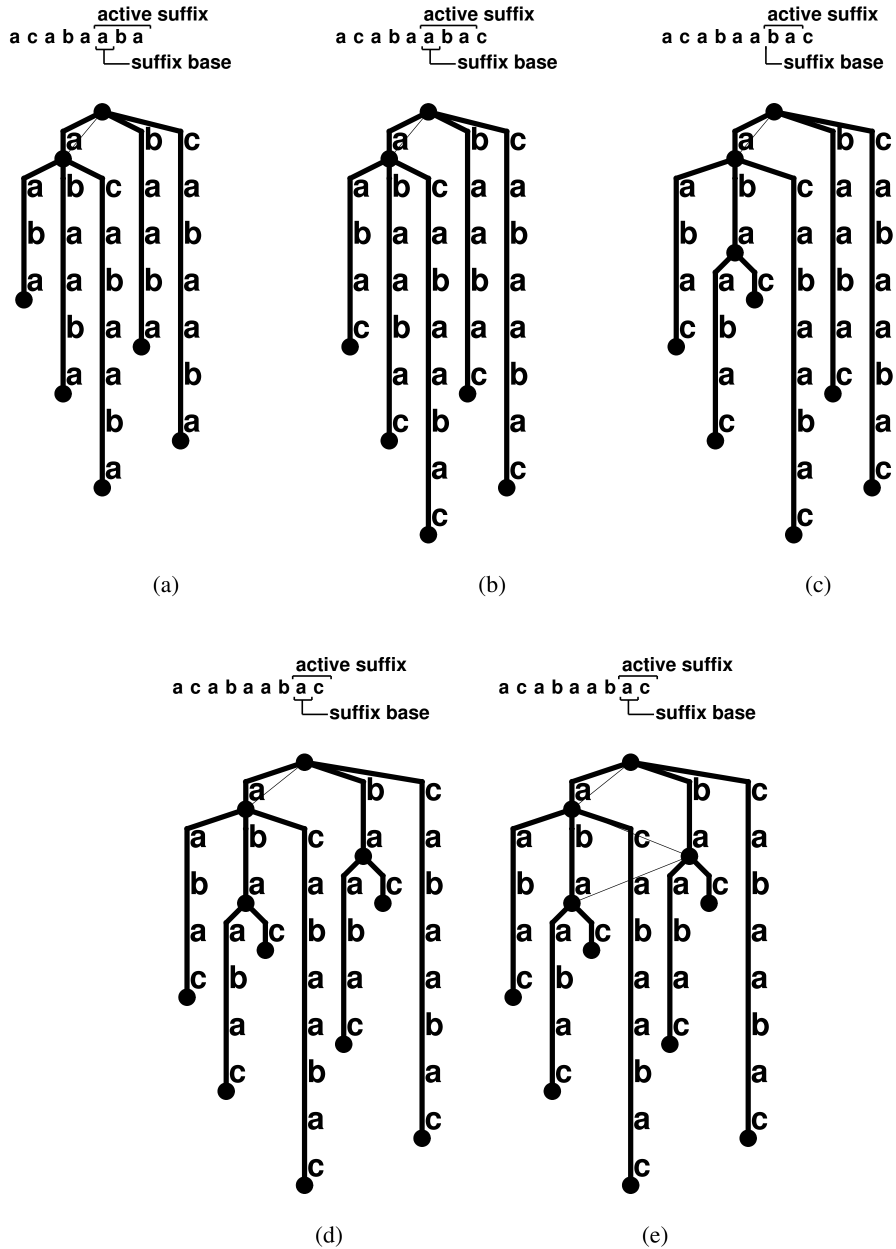


Figure 3.4: Steps of constructing $\mathrm{ST}(\mathtt{acabaabac})$ from $\mathrm{ST}(\mathtt{acabaaba})$. The active suffix (as represented by asp_{suffix}) is shown above each tree. The substring marked as suffix base is the part of the active suffix that is represented by the base of the (canonical) reference pair. The offset index of the reference pair points to the character one right of the substring represented by the base. Figure (a) is the starting point. In Figure (b) the open edges have grown with the text. Figure (c) shows the tree after the first relevant suffix has been inserted and the base of the reference pair has been moved over a suffix link. Figure (d) shows the tree after the second relevant suffix has been inserted, the reference pair has been shortened at the front, canonized to $\overline{\mathtt{a}}$, and was enlarged by the new character. Figure (e) shows the final tree with the updated suffix links.

Procedure 3.5 decanonize(*reference pair* $(p,(o,l))$, *character* a)

Parameters: $(p,(o,l))$ is a reference pair that is to be decanonized, a is the character for that a prefix edge is to be found.

1: $e := \text{getEdge}(p, \mathsf{PREFIX}, a)$; {If there is no edge,then e is nil.}
2: **while** $e = nil$ **or** $\text{getLength}(e) \neq 1$ **do**
3: $f := \text{getParent}(p, \mathsf{SUFFIX})$;
4: **if** $f = nil$ **then** {If p has no parent, it is the root. The index o should then point to the beginning of the old string. Make it point to the beginning of the new string.}
5: $o := o - 1$;
6: $l := l + 1$;
7: $\mathbf{return}$;
8: **else**
9: $k := \text{getLength}(f)$;
10: $o := o - k$;
11: $l := l + k$;
12: $p := \text{getSourceNode}(f)$;
13: $e := \text{getEdge}(p, \mathsf{PREFIX}, a)$;
14: $p := \text{getTargetNode}(e)$;

Proof. Obviously, both strings are prefixes of at. Either $\hat{\alpha}(at)$ is a prefix of $a\hat{\alpha}(t)$, or $a\hat{\alpha}(t)$ is a proper prefix of $\hat{\alpha}(at)$. If $a\hat{\alpha}(t)$ were a proper prefix of $\hat{\alpha}(at)$, then we would have $\hat{\alpha}(at) = a\hat{\alpha}(t)v$ for some $v \in \Sigma^+$. Hence, $\hat{\alpha}(t)v \in \text{prefixes}(t)$ and $a\hat{\alpha}(t)v \in \text{psubstrings}(t) \cup \text{suffixes}(t)$, which contradicts the definition of the active prefix because $\hat{\alpha}(t)v$ would be a longer candidate. □

Let $t = t_{-n} \cdots t_{-1}$ and $at = t_{-n-1}t_{-n} \cdots t_{-1}$ (i.e., $a = t_{-n-1}$). For the purpose of constructing the suffix tree $\text{ST}(at)$ from the suffix tree $\text{ST}(t)$, we keep a canonical reference pair $app_{\mathsf{suffix}}(t)$ for the active prefix $\hat{\alpha}(t)$. Let $r_{n+1} = (p_{n+1}, v_{n+1})$ be a canonical reference pair for $\hat{\alpha}(at) = t_{-1-n,i_{n+1}}$ in $\text{ST}(t)$ (the location of the new active prefix in the current suffix tree). The base p_{n+1} of r_{n+1} has a suffix link to a node p'_{n+1} (Lemma 3.2). Hence, $r' = (p'_{n+1}, v_{n+1})$ is a reference pair for $t_{-n,i_{n+1}}$, which is a prefix of $\hat{\alpha}(t)$ (Lemma 3.8). Let $r_n = (p_n, v_n)$ be the reference pair $app_{\mathsf{suffix}}(t)$. There exists a path $\pi = (p'_{n+1}, \ldots, p_n)$ from p'_{n+1} to p_n in $\text{ST}(t)$ (π might have length 0).

We claim that for any node s on the path π, if $s \neq p'_{n+1}$ and $s \neq p_n$, then there is no node q such that there is a suffix link labeled a from q to s.

Assume otherwise that such a node q existed. Because $\hat{\alpha}(t)$ is a prefix of t, $\text{path}(s)$ is also a prefix of t and $\text{path}(q) = a\text{path}(s)$ is a prefix of at. On the other hand, q being a leaf proves the existence of $\text{path}(q) \in \text{suffixes}(t)$ and q being an inner node proves the existence of $\text{path}(q) \in \text{prefixes}(t) \cup \text{psubstrings}(t)$. Hence, we have $\text{path}(q) \in \text{psubstrings}(at) \cup \text{suffixes}(at)$. Therefore, $\text{path}(q)$ is a nested-prefix of at and thus a prefix of $\hat{\alpha}(at)$. Because $\text{depth}(q) = \text{depth}(s) + 1 > \text{depth}(p'_{n+1})$, this is a contradiction to $r_{n+1} = (p_{n+1}, v_{n+1})$ being a canonical reference pair for $\hat{\alpha}(at)$.

As a result, we have a unique method to find $app_{\mathsf{suffix}}(at)$ from $app_{\mathsf{suffix}}(t)$ by

Procedure 3.6 update-new-suffix(*reference pair* $(p, (o, l))$, *character* a, *int* k)

Parameters: k is the new length of the reference pair after it has been decanonized.

1: $\text{decanonize}((p, (o, l)), a)$; {We remember node r at depth $k - 1$ during canonizing.}
2: $l := k - \text{depth}(q)$
3: **if** $l = 0$ **then**
4: $q := p$;
5: **else**
6: insert a node q at the location of $(p, (o, l))$ by splitting the edge with starting character t_o
7: insert a new leaf r linked to q by an edge with string label $(o, n + 1)$
8: **if** q was a leaf **then**
9: eliminate q
10: **else**
11: add a suffix link from q to r

decanonizing $app_{\mathsf{suffix}}(t) = (p, (i, l))$. We start at the base p and walk up towards the root until a prefix edge (i.e., a reverse suffix link) e labeled with a is found. For each edge on the way up, we subtract its length from i. Finally, we replace p by the target q of the prefix edge e (i.e., the source of the suffix link). With the knowledge of the length of $\hat{\alpha}(at)$, we can set the length l to $|\hat{\alpha}(at)| - \text{depth}(q)$ and get a canonical reference pair for $app_{\mathsf{suffix}}(at)$.

If a new node q is added for $app_{\mathsf{suffix}}(at)$ to insert the leaf representing the suffix at, we have to set its suffix link to a node r. By Lemma 3.8, we have $\text{depth}(r) \leq |\hat{\alpha}(t)|$. At the beginning of the iteration, we have the canonical reference pair $r_n = (p_n, v_n)$ for $\hat{\alpha}(t)$. Because r_n is canonical, we know that $\text{depth}(r) \leq \text{depth}(p_n)$. On the other hand, we inserted a new node q above p_{n+1}, thus we have $\text{depth}(r) = \text{depth}(q) - 1 > \text{depth}(p_{n+1}) - 1 = \text{depth}(p'_{n+1})$. Hence, we encounter r on the path π while searching the suffix link labeled a. We know beforehand that $\text{depth}(r) = |\hat{\alpha}(at)| - 1$, so r is also easily identified. Procedure 3.6 gives the algorithm in pseudo code and Procedure 3.5 the details of decanonizing.

Lemma 3.9.
The amount of work performed during all calls to $\text{decanonize}()$ *with the reference pair* $app_{\mathsf{suffix}}(t)$ *is* $O(|t|)$.

Proof. We apply amortized analysis as follows: Each time a character is prepended to t a coin is put on the character. As described above, for each edge the base is moved upwards, the index o of the string part of the reference pair is decreased. We take the coin on t_o to pay for the step. The offset o stays unchanged when the base is replaced or when the length is adjusted. If $\text{decanonize}()$ moves the base of $app_{\mathsf{suffix}}(t) = (p, (o, l))$ to a leaf, l must be zero (otherwise there wouldn't be a location for $app_{\mathsf{suffix}}(t)$). The leaf is deleted when attaching the new suffix to it and the base of $app_{\mathsf{suffix}}(t)$ is moved up towards the next node, decreasing the offset o. Therefore, o never increased and there is always a coin on t_o. As a result, decanonizing has

amortized constant complexity. □

Theorem 3.10.
With the additional information of knowing the length of $\hat{\alpha}(s)$ for any suffix s of t before inserting it, it takes time $O(|t|)$ to construct $\mathrm{ST}(t)$ in an anti-on-line manner, i.e., reading t in reverse direction from right to left.

Proof. Inserting the leaf, inserting an inner branching node, or deleting an old leaf takes constant time per step. The non-constant part of each iteration is the decanonizing of $app_{\mathsf{suffix}}(t)$. By Lemma 3.9, the work can be done in amortized constant time. □

See Figure 3.7, where the construction of $\mathrm{ST}(\mathtt{cabaabaca})$ from $\mathrm{ST}(\mathtt{abaabaca})$ is shown. If we do not have prefix links for leaves, the reference pair for $app_{\mathsf{suffix}}(t)$ is decanonized by moving the base from $\overline{\mathtt{aba}}$ over $\overline{\mathtt{a}}$ to the root, where the offset is decreased by one. The length is then adjusted to 2. The leaf will be deleted and, therefore, there is no need to canonize to $\overline{\mathtt{ca}}$. If we have prefix links for leaves (not drawn in Figure 3.7), there is a prefix edge labeled $\mathtt{c}$ from $\overline{\mathtt{a}}$ to $\overline{\mathtt{ca}}$, which is found after moving the base up to $\overline{\mathtt{a}}$. Before deleting the node $\overline{\mathtt{ca}}$, the base of the reference pair needs to be moved back up to the root.

3.3 Constructing Compact Affix Trees On-Line

This section describes the linear-time unidirectional construction of affix trees. We emphasize the derivation from merging the two algorithms of the previous section. The algorithm is best understood as interleaving the steps of these algorithms together with the use of paths so that the two algorithms never notice each other.

3.3.1 Overview

The algorithm is dual by design. The necessary steps to construct $\mathrm{AT}(ta)$ from $\mathrm{AT}(t)$ are the same as for constructing $\mathrm{AT}(at)$ from $\mathrm{AT}(t)$, we only have to exchange "suffix" and "prefix" in the description and swap the reference pairs. For the sake of clarity, we assume the view of constructing $\mathrm{AT}(ta)$ from $\mathrm{AT}(t)$. We call this a *suffix iteration*, whereas constructing $\mathrm{AT}(at)$ from $\mathrm{AT}(t)$ is called a *prefix iteration*.

The affix tree for t consists of the suffix structure that is equivalent to the suffix tree $\mathrm{ST}(t)$ and the prefix structure that is equivalent to the prefix tree $\mathrm{ST}(t^R)$. To create the affix tree $\mathrm{AT}(ta)$ from the affix tree $\mathrm{AT}(t)$, we apply the algorithm described in Section 3.2.1 (Ukkonen) to the suffix structure and apply the algorithm described in Section 3.2.2 (Weiner′) to the prefix structure. We need to take precaution so that one algorithm does not disturb the other when they are merged together (the algorithms need to "see" the correct view). Additionally, the path structures must be updated correctly.

To achieve a linear running time, we treat paths as edges. Function 3.8 shows how prefix nodes clustered in a path can be treated as a single edge (in constant time). All other functions on edges can be implemented for paths in a similar way with constant running times.

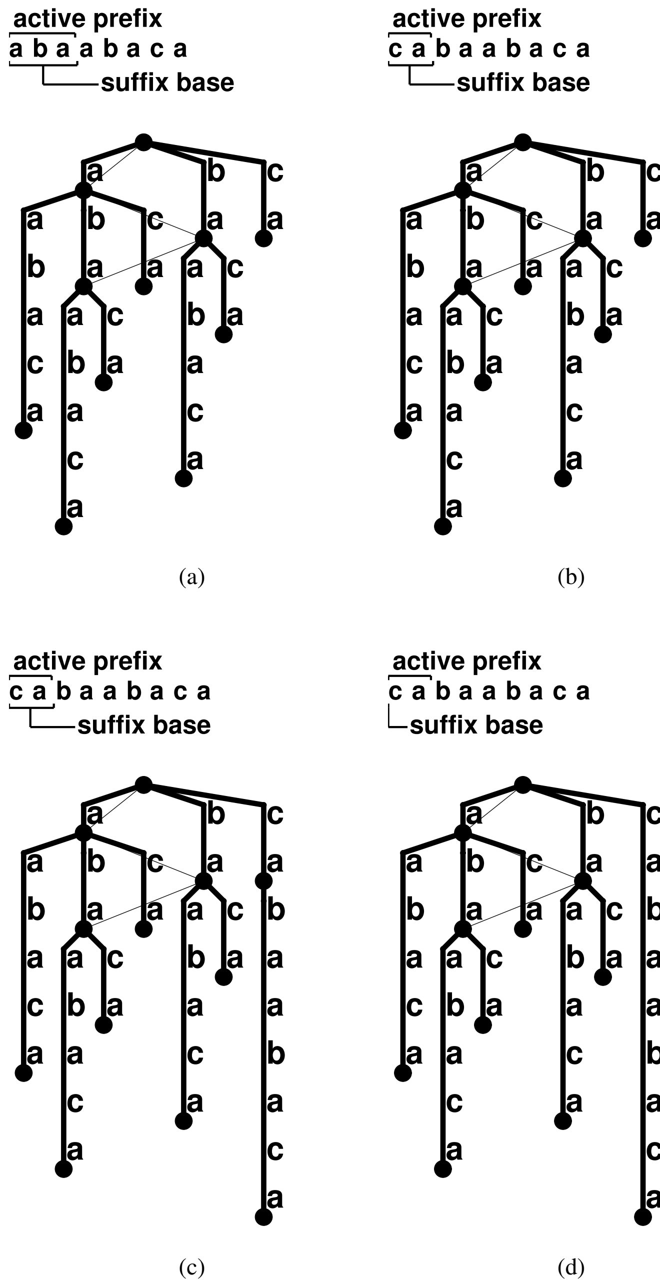


Figure 3.7: Steps of constructing $\mathrm{ST}(\texttt{cabaabaca})$ from $\mathrm{ST}(\texttt{abaabaca})$. The active prefix (as represented by app_{suffix}) is shown above each tree. The substring marked as prefix base is the part of the active prefix that is represented by the base of the (canonical) reference pair. The offset index of the reference pair points to the character one right of the substring represented by the base. Figure (a) is the starting point. In Figure (b), the app_{suffix} has been decanonized and the location of $\hat{\alpha}(\texttt{cabaabaca})$ has been found. Figure (c) shows the tree with the new suffix inserted. It was added at a leaf, which is then deleted to give the final tree in Figure (d).

Function 3.8 $\text{getTargetNodeVirtualEdge}(\textit{edge } e)$ **returns** $node$

Parameters: e is a suffix edge that possibly ends in a prefix node.

1: $p := \text{getTargetNode}(e)$;
2: **if** p has only one suffix child **then** {Node p is a prefix node.}
3: $\pi = \text{getPath}(p)$; {Node p is contained in a path π.}
4: $q = \text{getLast}(\pi)$; {Node q is the last node of path π.}
5: $f = \text{getOnlyEdge}(q, \mathsf{SUFFIX})$; {Node q is a prefix node (in π) and has only one suffix child.}
6: $r = \text{getTargetNode}(f)$;
7: $\mathbf{return}(r)$;
8: **else**
9: $\mathbf{return}(\text{p})$; {There is no path for suffix nodes.}

Each iteration (which consists of all necessary updating steps to append a character to t) of the algorithms consists of the following **steps**:

Step 1. Let p be the prefix parent of $asl_{\mathsf{suffix}}(t)$. Remove the prefix edge that leads from p to $asl_{\mathsf{suffix}}(t)$. Remember the node p.

Step 2. Lengthen the text, thereby automatically extending all open edges.

Step 3. Insert a prefix node for t as suffix parent of $\overline{ta}$.

Step 4. Insert the relevant suffixes and update the suffix links using $asp_{\mathsf{suffix}}(t)$. Restore the prefix structure to include the prefix t.

Step 5. Decanonize $asp_{\mathsf{prefix}}(t)$ to find $asp_{\mathsf{prefix}}(ta)$, make its location explicit, and add a prefix edge to the current asl_{suffix}. If the location of $asp_{\mathsf{prefix}}(ta)$ is a prefix leaf, then the leaf must be deleted because it has become a non-branching inner node in this step.

3.3.2 Detailed Description

Step 1 is necessary because the suffix edge is an open edge and automatically grows when the string is enlarged, while the prefix edge is not an open edge. After this step, the prefix structure is still complete except for the prefix for t, which is no longer represented. The step can easily be implemented in constant time.

The underlying string can be implemented as a dynamically growing array (see Section 2.2.2). When a character is appended in **Step 2**, all open edges and thus all suffixes of Type 1 (see Section 3.2.1) grow automatically. The prefix link of the active suffix leaf was deleted in the previous step because otherwise $\text{depth}(asl_{\mathsf{suffix}})$ would be different in the prefix and in the suffix part of the affix tree.

Step 3 reinserts the prefix node for t (but without linking it in the prefix structure yet). The prefix edge will be inserted in the next step. We can find the correct location to insert the new node easily if we keep a reference to the node that represents the complete string (the *end node*). The node will never change and automatically grows

by its open edge. It can never be deleted because in Step 5 leaves are only deleted if they become nested. The new prefix node for t is not a suffix node (it cannot be branching and it is not a leaf) and is therefore added to a path. If t is not the only prefix leaf representing a proper non-nested prefix of the complete string, there is a prefix leaf r representing $t_{0,n-1}$ and the new leaf can be added to r's path.

Step 4 is basically the same as the algorithm described in Section 3.2.1. For each new suffix leaf, a prefix edge to the current asl_{suffix} is inserted and the new leaf becomes the asl_{suffix}.

Although the location referenced by asp_{suffix} may be implicit in the suffix structure, each time a new node would be inserted for a suffix tree, there can be an existing node p that represents a left-branching substring of t and belongs to the prefix structure. In this case, the path that contains p must be split and p must thus be made visible in the suffix structure. This corresponds to splitting an edge and can be implemented in constant time analogously to Function 3.8.

The newly inserted inner nodes must be threaded in the prefix structure which corresponds to setting their suffix links. By Lemma 3.2, there is always an end point (the suffix parent to the last node where a leaf was added), which can be used as a starting point. If new nodes were inserted, these belong to the suffix structure only and the corresponding paths must be added or changed.

The procedure $\text{update-relevant-suffixes}(asp_{\mathsf{suffix}}, a)$ is modified to put the traversed nodes on a stack, and their suffix links are set in reverse order while the paths are augmented at the same time. By traversing the nodes in reverse order, we can assure that we are able to grow existing paths. When setting the suffix links and adding nodes to paths, we must check whether old prefix edges between these nodes existed and if so remove them. This can be done while the nodes are put on the stack. The next two lemmas give some hints on how to reconstruct the prefix structure.

Lemma 3.11.
Let p be a (new) branching node to that a relevant suffix is attached. Then p is a prefix parent or prefix ancestor of the prefix leaf representing t.

Proof. Let p be as in the prerequisite. Then p represents a string $w = \text{path}(p)$ that was a nested suffix of t, but wa is not nested in ta. Because w is a suffix of t, there is a prefix path from p to t. □

Thus, all inserted nodes represent suffixes of t.

Lemma 3.12.
The string $\text{path}(p)$ represented by the node p remembered in Step 1 is a suffix of $\alpha(t)$.

Proof. The node p is the prefix parent of $asl_{\mathsf{suffix}}(t)$. Therefore, $\text{path}(p)$ is a suffix of $\text{path}(asl_{\mathsf{suffix}}(t))$. Because $\text{path}(asl_{\mathsf{suffix}}(t))$ is a suffix of t, so is $\text{path}(p)$. The node p is either an inner prefix node or an inner suffix node. In both cases, it represents a nested substring of t (by Lemma 3.1). We have $|\text{path}(p)| \leq |\alpha(t)|$ because $\alpha(t)$ is the largest nested suffix of t. Hence, $\text{path}(p)$ is a suffix of $\alpha(t)$. □

Let q be the first node that was either inserted or that resulted from a split path (in which case q was a prefix but no suffix node in $\text{AT}(t)$). Let p be the node remembered in Step 1. Note that all nodes inserted after node q represent suffixes of $\text{path}(q)$,

and $\mathrm{path}(q)$ is a suffix of t. Furthermore, $\mathrm{path}(p)$ is nested in t and thus shorter than $\mathrm{path}(q)$, which is also a suffix of t. Because p was the direct prefix parent of $asl_{\mathsf{suffix}}(t)$, there is no other suffix of t represented by a node in $\mathrm{AT}(t)$. It follows that q (representing $\alpha(t)$) is either a descendant of p or equals p and that we have thus already inserted a chain of suffix links from q to p. Hence, we insert a prefix edge from q to the node inserted for t in Step 3. If no relevant suffixes were inserted, we add a prefix edge from node p to the prefix node for t.

While the prefix t was removed from the prefix structure in Step 1, it is now inserted again. The newly inserted inner nodes are in the suffix structure only and are hidden in paths. After this step, the prefix structure corresponds to $\mathrm{ST}(t^R)$ again.

Step 5 inserts the prefix ta and basically corresponds to the algorithm described in Section 3.2.2. The suffix location is easily found with $asp_{\mathsf{suffix}}(ta)$ and so the node can be inserted in the suffix structure, too (possibly in a path). The length of $\hat{\alpha}(ta)$ is the same as the length of $\alpha(ta)$ and the information is taken from the previous step.

If a new prefix node is inserted, it may either be added to a path on its own or to the path of its suffix parent. If the location is a prefix leaf, it must be removed from the path that contains all the leaves before it is deleted.

All suffix leaves are suffixes of ta. They are connected by prefix edges labeled with the first characters of ta. For example, the largest suffix is $t_0 \cdots t_n a$, the second largest suffix is $t_1 \cdots t_n a$. They are connected by a prefix edge labeled with t_0. Hence, the prefix edges connecting the $k+1$ largest suffix leaves describe the prefix $t_0 \cdots t_k$. All leaves but the leaf for ta are in a path and not visible in the prefix tree. They are the end of the largest prefix ta. Linking asl_{suffix}, the shortest suffix leaf, to the location $asp_{\mathsf{prefix}}(ta)$ is equivalent to inserting a new prefix leaf for ta.

Hence, Step 5 correctly updates the prefix structure of the affix tree and the resulting tree is $\mathrm{AT}(ta)$.

3.3.3 An Example Iteration

Figures 3.9 and 3.10 show an example iteration of building $\mathrm{AT}(\texttt{acabaabac})$ from $\mathrm{AT}(\texttt{acabaaba})$. Figure 3.9 shows the suffix view, and Figure 3.10 shows the prefix view. The trees correspond to each other. Figure (a) shows the affix tree for the string acabaaba. After Step 1 the prefix link of asl_{suffix} (in the suffix view $\overline{\texttt{aaba}}$) to p (in the suffix view $\overline{\texttt{aba}}$) is removed. The prefix for $t = \texttt{acabaaba}$ is no longer represented anymore, which is shown in Figure (b). Figure (c) shows the tree after the string has grown in Step 2. As the character was appended to the right, this has no effect on the prefix structure while all suffix leaves grow by the open edges. Step 3 reinserts the node for the prefix location of t. The node is added to the path that contains the prefix leaves as shown in Figure (d). The first relevant suffix is inserted resulting in Figure (e). The branching node $q = \overline{\texttt{aba}}$ would have been inserted in a suffix tree construction. In the affix tree, the node already exists and is only made a suffix node by removing it from the path. This has no effect on the prefix structure. Figure (f) shows the tree after the second relevant suffix has been inserted. A new branching node $q = \overline{\texttt{ba}}$ was created in the suffix structure. The new node is not a prefix node and will thus be added to a path. It appears—hidden in a path—in the prefix structure after the new nodes have been threaded in from the stack. This can be seen in Figure (g), which shows the tree after Step 4. The reference pair asp_{prefix} is decanonized, its base moves from $\overline{\texttt{aba}}$ to $\overline{\texttt{a}}$,

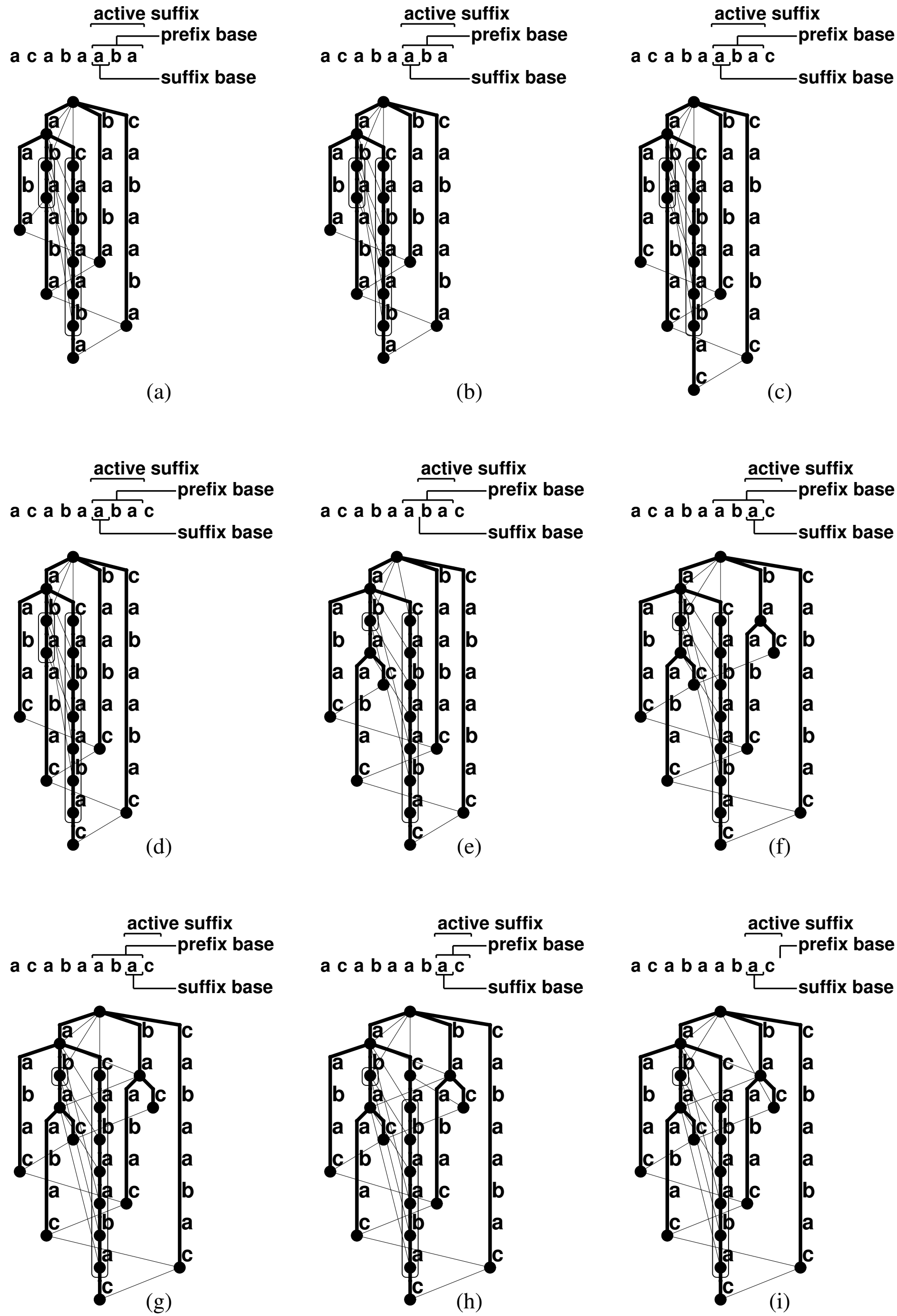


Figure 3.9: Steps of an iteration from $\mathrm{AT}(\texttt{acabaaba})$ to $\mathrm{AT}(\texttt{acabaabac})$ in the suffix view. The figures show the affix tree (a) $\mathrm{AT}(\texttt{acabaaba})$, (b) after Step 1, (c) after Step 2, (d) after Step 3, (e) after inserting the first relevant suffix, (f) after inserting the second relevant suffix, (g) after Step 4, (h) after decanonizing asp_{prefix} and inserting the prefix location of ta, and (i) after the prefix leaf has been deleted. The resulting tree is $\mathrm{AT}(\texttt{acabaabac})$.

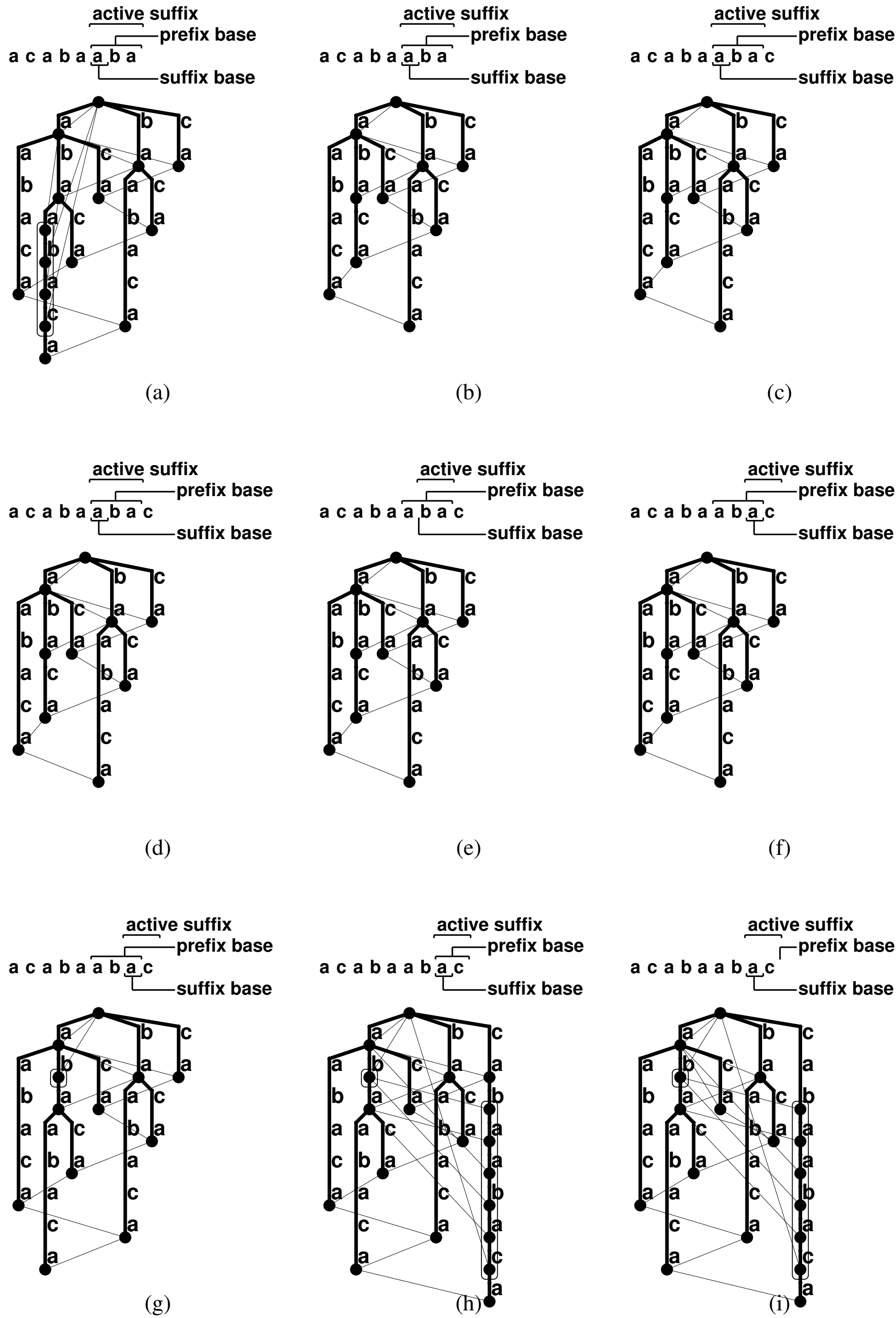


Figure 3.10: Steps of an iteration from $\mathrm{AT}(\mathtt{acabaaba})$ to $\mathrm{AT}(\mathtt{acabaabac})$ in the prefix view. The figures show the affix tree (a) $\mathrm{AT}(\mathtt{acabaaba})$, (b) after Step 1, (c) after Step 2, (d) after Step 3, (e) after inserting the first relevant suffix, (f) after inserting the second relevant suffix, (g) after Step 4, (h) after decanonizing asp_{prefix} and inserting the prefix location of ta, and (i) after the prefix leaf has been deleted. The resulting tree is $\mathrm{AT}(\mathtt{acabaabac})$.

where a suffix edge leads to $\overline{\texttt{ca}}$, which becomes the new base. The new length is set to two, and a prefix edge is inserted to $asl_{\mathsf{suffix}} = \overline{\texttt{cab}}$. The resulting tree is shown in Figure (h). Because $\overline{\texttt{ca}}$ was a prefix leaf, it needs to be deleted. Finally, the resulting tree $\mathrm{AT}(\texttt{acabaabac})$ is shown in Figure (i).

3.3.4 Complexity

Theorem 3.13 (Complexity of the unidirectional construction).
The affix tree $\mathrm{AT}(t)$ *for the string* t *can be constructed in an on-line manner from left to right or from right to left in time* $O(|t|)$.

Proof. The algorithm is dual, so it does not matter whether we always append to the right or always append to the left. Exchanging the words "suffix" and "prefix" in the description of the algorithm does not change it.

With the use of paths (see Function 3.8 for an example), Step 4 only sees nodes that would be in the corresponding suffix tree. The other nodes are hidden in paths and are skipped like edges. Similarly, Step 5 only works with nodes that would be in the corresponding prefix tree. New nodes are added to paths and so this invariant is kept. Therefore, Lemma 3.6 and Lemma 3.9 can be applied and both steps take amortized constant time. If we use the index transformation as described in Section 3.1.3, we do not even need to change the amortized analysis of Lemma 3.9—the coins are simply taken from t_{-1-o} (recall that o is the offset of the reference pair asp_{prefix}). Because o only decreases, $-1-o$ only increases. We always put a coin on a newly added character. Hence, there is always a coin on t_{-1-o}. The other three Steps (1, 2, and 3) take constant time each. As a result, the algorithm runs in linear time. □

Remark 3.14.
As already stated, the algorithm is on-line: The k-th step results in the affix tree for the prefix of length k of the string. Assuming the input string has length n, the complexity of an individual step can well be $\Theta(n)$. Consider the last step of building the affix tree for $\texttt{aaa}\cdots\texttt{aaab}$. The string $\texttt{aaa}\cdots\texttt{aaa}$ has no right- or left-branching substrings, but the final string has $n-2$ right-branching substrings, thus $\Theta(n)$ new inner nodes have to be created.

3.4 Bidirectional Construction of Compact Affix Trees

So far, we have only considered the question whether it is at all possible to construct an affix tree in linear time. We answered the question in the positive. We even have an on-line algorithm with amortized constant cost per step that constructs a completely dual data structure. This naturally raises the question what happens if the on-line construction is done in both directions. This bidirectional construction is on-line by adding characters at both sides of the string in arbitrary order. Besides the need to update the important locations for both sides (e.g., the reference pair for the active prefix point), additional measures must be taken to achieve a linear running time. Lemmas 3.15 and 3.16 can be found in a similar form in [Sto95] and are replicated for a better understanding, clarity, and completeness. The importance of Lemma 3.16 reveals itself only in the context of the amortized analysis for the linear bidirectional construction of compact affix trees.

3.4.1 Additional Steps

The affix tree construction algorithm is dual, so the tree can be constructed by appending characters at the end or at the front of the underlying string. To be able to do this interchangeably, we need to remember the reference pairs for the active prefix point (app_{prefix} and app_{suffix}), too, and we need to keep them canonical and up-to-date. For this, we add a **new step** to the steps described in Section 3.3:

Step 6. Update the active prefix.

By appending a character a to the end of t the active prefix $\hat{\alpha}(\cdot)$ cannot become smaller. The active prefix $\hat{\alpha}(t)$ stays a prefix in ta and, if it is nested in t, it is also nested in ta. Therefore, the active prefix can grow at most by one character if it is a prefix and a suffix of t and the prefix is followed by a, i.e., $\hat{\alpha}(t)a \in \mathrm{prefixes}(t)$ and $\hat{\alpha}(t) \in \mathrm{suffixes}(t)$. The following lemma can be used to determine if the active prefix grows.

Lemma 3.15.
The active prefix grows by one character if and only if the prefix location of the new active suffix is represented by a prefix leaf in Step 5 of the algorithm.

Proof. If the prefix location of the new active suffix is a prefix leaf, then the new active suffix is also a prefix of ta. We have $\alpha(ta) \in \mathrm{prefixes}(ta)$ and $\alpha(ta) \in \mathrm{suffixes}(ta)$. Obviously, $\alpha(ta)$ is a nested prefix. Because $v = \alpha(ta)$ is represented by a prefix leaf in $\mathrm{AT}(t)$, the string v was a non-nested prefix of t before. Therefore, the active prefix $\hat{\alpha}(t)$ was smaller than v. Due to the fact that v is a nested prefix of ta, the new active prefix $\hat{\alpha}(ta)$ must be at least as large as v. We only add one character to t, thus all new substrings of ta not in t are substrings of t appended by a. Therefore, the active prefix cannot grow by more than a single character.

For the other direction of this proof, assume that the active prefix grows by one character. Then its new larger second occurrence in ta must be a suffix. Hence, we have $\hat{\alpha}(ta) \in \mathrm{prefixes}(ta)$ and $\hat{\alpha}(ta) \in \mathrm{suffixes}(ta)$. The new active prefix $\hat{\alpha}(ta)$ is a nested suffix of ta and therefore a suffix of $\alpha(ta)$. We claim that $\hat{\alpha}(ta) = \alpha(ta)$. For the sake of contradiction, suppose that the active suffix is larger, then we have $v\hat{\alpha}(ta) = \alpha(ta)$ for some $v \in \Sigma^+$. There must be a second occurrence of the active suffix $v\hat{\alpha}(ta)$ in ta that is not a suffix. If $v\hat{\alpha}(ta)$ is also a prefix, then it is a candidate for the active suffix contradicting our assumptions about $\hat{\alpha}(ta)$. Therefore, $v\hat{\alpha}(ta)$ is a proper substring of ta and consequently also a non-prefix substring of t. The situation is described by

$$\hat{\alpha}\,(ta) \in \mathrm{prefixes}\,(t) \text{ and } \hat{\alpha}\,(ta) \in \mathrm{psubstrings}\,(t) \cup \mathrm{suffixes}\,(t) \quad . \tag{3.6}$$

Hence, $\hat{\alpha}(ta)$ is a nested prefix of t and, as such, a candidate for the active prefix of t larger than the active prefix itself—a contradiction. Therefore, we must have $\hat{\alpha}(ta) = \alpha(ta)$. Because the new active prefix is longer than the old one, $\hat{\alpha}(ta)$ is a non-nested prefix in t, and it is represented by a prefix leaf in $\mathrm{AT}(t)$. □

We find the prefix location of the new active suffix in Step 5. We then check explicitly whether it is a prefix leaf (and whether we must delete it) or not. Step 6 is

easily implemented by canonizing both reference pairs if no node is deleted in Step 5 (we canonize because nodes might have been inserted), and by lengthening the active prefix when a leaf is deleted. We now detail the lengthening step.

If the active prefix grows, we must keep its reference pairs up-to-date. In that case, we also know that $\hat{\alpha}(ta) = \hat{\alpha}(t)a$. To find the prefix location $app_{\mathsf{prefix}}(ta)$ of $\hat{\alpha}(ta)$ from $app_{\mathsf{prefix}}(t)$, we must find a suffix edge labeled a to move its base so that the new character can be prepended. This can be done with the already known procedure $\mathrm{decanonize}(asp_{\mathsf{prefix}}(t), a)$. The location of $app_{\mathsf{suffix}}(ta)$ can be found by lengthening $app_{\mathsf{suffix}}(t)$ by the character a. After that, we use $\mathrm{canonize}(app_{\mathsf{suffix}}(ta))$ to keep the reference pair canonical.

If the active prefix does not grow, it is still necessary to canonize $app_{\mathsf{prefix}}(t)$ and $app_{\mathsf{suffix}}(t)$ because prefix and suffix nodes might have been inserted. Calling canonize at the end of an iteration is equivalent to simply testing whether a newly inserted suffix (prefix) node is the new canonical base of app_{suffix} (asp_{prefix}) each time a new node is added. This adds only constant overhead to the insertion of nodes. Step 6 hence adds the additional overhead of canonizing $app_{\mathsf{suffix}}(t)$ and decanonizing $asp_{\mathsf{prefix}}(t)$.

Thus, we can construct affix trees bidirectional and on-line. To achieve a linear-time behavior, we need to apply some more changes as explained next.

3.4.2 Improving the Running Time

If the algorithm is executed as described above, a linear time bound cannot be guaranteed if characters are appended and prepended in arbitrary order. As an illustration, suppose that after some iterations where characters were appended to the string, characters are now prepended. In such a "reverse" iteration, the active suffix might grow. In Step 6 of the reverse iteration, the reference pair $asp_{\mathsf{suffix}} = (p, (o, l))$ is decanonized. Thus, its index o into the string is decreased and Lemma 3.6 cannot be applied anymore. Similar conditions may increase the index o of asp_{prefix} and Lemma 3.9 cannot be applied.

To prevent costly oscillation of the indices and to reduce work to amortized linear time, we must conserve some work. An amortized analysis will then result in the desired complexity. We introduce two improvements to Step 5 and to Step 6 that ensure a linear time bound.

Extension 1.
If the new prefix location of the active suffix is already explicit as a prefix node q, this node must occur in a path starting below the base p of the $asp_{\mathsf{suffix}} = (p, (o, l))$. We can get to the node in constant time by retrieving the $(l-1)$-th element in the path starting at the child t_o of p. In Step 5, we decanonize only if the node cannot be found in constant time by the method described above.

In the example shown in Figures 3.9 and 3.10, the canonical suffix reference pair $asp_{\mathsf{suffix}}(ta)$ after Step 4 (see Figure (g)) has the base $\bar{\mathtt{a}}$ and the length 1. This location is represented by the prefix node $\overline{\mathtt{ca}}$ in the prefix structure and can be retrieved in constant time from node $\bar{\mathtt{a}}$.

The following lemma will make Step 6 run in constant time altogether.

Lemma 3.16.
If the active prefix $\hat{\alpha}(t)$ grows, the new active prefix $\hat{\alpha}(ta)$ is equal to the new active suffix $\alpha(ta)$.

Proof. If the active prefix grows from t to ta, then $\hat{\alpha}(t)a$ must be a prefix and $\hat{\alpha}(t)$ a suffix of the original string, and there may not be any other occurrence of $\hat{\alpha}(t)a$ anywhere in t (otherwise $\hat{\alpha}(t)a$ is a larger nested prefix—a contradiction to $\hat{\alpha}(t)$ being the active prefix of t). The new string ta has the suffix $\hat{\alpha}(t)a$ and the prefix $\hat{\alpha}(t)a$. Hence, $\hat{\alpha}(t)a$ is a nested suffix.

Suppose there is a larger nested suffix $w = v\hat{\alpha}(t)a$ of ta. Then w is a suffix of ta, and there must be an additional occurrence of w in ta, i.e., $w \in \text{prefixes}(ta) \cup \text{psubstrings}(ta)$. Hence, the suffix $\hat{\alpha}(t)a$ of w must be either a proper substring or a suffix of t. But then we have $\hat{\alpha}(t)a \in \text{prefixes}(t)$ and $\hat{\alpha}(t)a \in \text{psubstrings}(t) \cup \text{suffixes}(t)$, so $\hat{\alpha}(t)a$ is a candidate for the active prefix of t—a contradiction. As a result, $\hat{\alpha}(ta) = \hat{\alpha}(t)a = \alpha(ta)$. □

This yields our second improvement.

Extension 2.
If the active prefix grows, we assign $asp_{\mathsf{suffix}}(ta)$ to $app_{\mathsf{suffix}}(ta)$ and $asp_{\mathsf{prefix}}(ta)$ to $app_{\mathsf{prefix}}(ta)$. By Lemma 3.16, this is correct, and Step 6 thus runs in constant time.

3.4.3 Analysis of the Bidirectional Construction

By Extension 2, Step 6 runs in constant time. The cost of canonizing asp_{prefix} and app_{suffix} when the active prefix does not grow can simply be assigned to the insertion cost of a node at no further asymptotic charge (only one or two reference pairs need to be checked for each newly inserted node). Hence, all that is left to take care of are the side effects that a reverse iteration has on later iterations. Because the reference pairs are all kept up-to-date and the upkeeping of asp_{prefix} and app_{suffix} comes for free, the main concern lies on changes of the reference pairs for the active suffix in iterations where characters are appended to the front of the underlying string.

3.4.3.1 Overview

The bidirectional version of the algorithm has four steps that do not run in constant time. These are Steps 4 and 5 in the suffix and in the prefix iteration (the procedures $\text{canonize}()$ and $\text{decanonize}()$). We distinguish between a suffix iteration (a "normal iteration") where a character a is appended to t resulting in ta, and a prefix iteration (a "reverse iteration") where a character a is prepended to t resulting in at.

We need to show that all executions of $\text{canonize}()$ and of $\text{decanonize}()$ in the prefix and in the suffix iteration have constant amortized cost. To show this, we use the accounting method of amortized analysis and assign to each character of t four purses or accounts:

- A "suffix/canonize" purse,
- a "suffix/decanonize" purse,
- a "prefix/canonize" purse, and

- a "prefix/decanonize" purse.

In the suffix iterations, the cost of a repetition of the while loop in $\mathrm{canonize}()$ is payed from a "suffix/canonize" purse and the cost of a repetition of the while loop in $\mathrm{decanonize}()$ is payed from a "suffix/decanonize" purse. Analogously, in prefix iterations, the "prefix/canonize" and the "prefix/decanonize" purses are used. The purses are filled when characters are appended or prepended to the string or when new nodes are inserted into the tree (the details of the latter are given further below). The following invariants are kept throughout the algorithm. They guarantee a linear behavior.

Invariant I Whenever the body of the while loop of $\mathrm{canonize}()$ in a suffix iteration is executed, there is a coin in the "suffix/canonize" purse of character t_o, where o is the offset index of asp_{suffix}.

Invariant II Whenever the body of the while loop of $\mathrm{decanonize}()$ in a suffix iteration is executed, there is a coin in the "suffix/decanonize" purse of character t_o, where o is the offset index of asp_{prefix}.

Invariant III Whenever the body of the while loop of $\mathrm{canonize}()$ in a prefix iteration is executed, there is a coin in the "prefix/canonize" purse of character t_o, where o is the offset index of app_{prefix}.

Invariant IV Whenever the body of the while loop of $\mathrm{decanonize}()$ in a prefix iteration is executed, there is a coin in the "prefix/decanonize" purse of character t_o, where o is the offset index of app_{suffix}.

Theorem 3.17 (Complexity of bidirectional construction of affix trees).
If the Invariants I, II, III, and IV are true, then the bidirectional construction of the affix tree for the string t has linear running time $O(|t|)$.

Proof. Let $n = |t|$. If the above four conditions are met, then each call to one of the procedures has constant amortized cost. In each iteration, at most one call to $\mathrm{decanonize}()$ is made and for each suffix leaf inserted in a suffix iteration and each prefix leaf inserted in a prefix iteration at most one call is made to $\mathrm{canonize}()$. Because there are at most n iterations, no more than n leaves can be deleted. The final affix tree has at most n suffix leaves and at most n prefix leaves. Hence, the number of leaves ever inserted is $O(n)$ and there are no more than $O(n)$ calls to $\mathrm{canonize}()$. □

As a result of the above theorem, each iteration has constant amortized cost.

We show that, even though an arbitrary number of reverse iterations is executed between normal iterations, there is always a coin in the "suffix/canonize" purse of character t_o when the body of the while loop of $\mathrm{canonize}()$ in a suffix iteration is executed and there is always a coin in the "suffix/decanonize" purse of character t_o, when the body of while loop of $\mathrm{decanonize}()$ in a suffix iteration is executed (Invariants I and II). By the duality of the algorithm, this will show that all four invariants are true throughout the algorithm and that the algorithm is linear in the length $|t|$ of the final string t.

We initialize each purse as follows:

- If a character a is appended to t, then the new string is $t_0 \cdots t_n t_{n+1}$ (with $a = t_{n+1}$), and we put a coin in the "suffix/canonize" purse of t_{n+1} and in the "suffix/decanonize" purse of t_{n+1}.
- If a character a is prepended to t, then the new string is $t_{-1} t_0 \cdots t_n$ (with $a = t_{-1}$), and we put a coin in the "prefix/canonize" purse of t_{-1} and in the "prefix/decanonize" purse of t_{-1}.

If we solely perform suffix iterations, the offset index o of asp_{suffix} is only increased and the offset index o' of asp_{prefix} is only decreased (so that the transformed index $-1 - o'$ is only increased). If no reverse iterations are performed, only the Invariants I and II (or III and IV) are relevant, and Lemma 3.18 tells us, that they are always fulfilled. Therefore, the construction is linear in time.

Lemma 3.18.
If the offset index o of asp_{suffix} and of app_{prefix} increases only, Invariants I and III are always fulfilled.
If the offset index o' of asp_{prefix} and of app_{suffix} decreases only, Invariants II and IV are always fulfilled.

Proof. Characters appended to the underlying string appear at higher indices than the index offset o of asp_{suffix} and at smaller indices (in the reverse view of the string) than the index o' of app_{prefix}. Similarly, characters prepended to the underlying string appear at higher indices (in the reverse view of the string) than the index offset o of asp_{prefix} and at smaller indices than the index o' of app_{suffix}. □

If intermediate prefix iterations are performed, the offset indices of asp_{suffix} and app_{prefix} can decrease (respectively increase). We have to prove that the additional coins needed can be put into the purses with amortized constant extra cost. The remaining two sections will show how this is done and complete the proof.

3.4.3.2 Changes to asp_{suffix} in a Reverse Iteration—Invariant I

Changes to asp_{suffix} concern Invariant I only. The following results can be applied analogously to app_{prefix} (by exchanging "suffix" and "prefix") and Invariant III.

Lemma 3.19.
Assigning $app_{\mathsf{suffix}}(at)$ to $asp_{\mathsf{suffix}}(ta)$ in Step 6 adds amortized constant overhead to successive calls to $\mathrm{canonize}(asp_{\mathsf{suffix}})$ *and keeps Invariant I.*

Proof. In a prefix iteration $t = t_{-m} \cdots t_n$ to $at = t_{-m-1} \cdots t_n$ (where the character $t_{-m-1} = a$ was prepended), the active suffix may grow by one. Although $asp_{\mathsf{suffix}}(t)$ is not decanonized explicitly (Extension 2), it is changed as if it had been decanonized. Let p^{suff}_{new} be the new base of $asp_{\mathsf{suffix}}(at)$ and let $p^{\mathsf{suff}}_{\mathsf{old}}$ be the old base, then there is a suffix node q such that q is the prefix parent of p^{suff}_{new} (the prefix edge is labeled by the prepended character a), and $p^{\mathsf{suff}}_{\mathsf{old}}$ is a suffix descendant of q. (See Figure 3.10(g), where the active prefix grows from $\mathtt{a}$ to $\mathtt{ac}$. We have $p^{\mathsf{suff}}_{\mathsf{old}} = \overline{\mathtt{aba}}$, $q = \mathrm{root}$, and $p^{\mathsf{suff}}_{new} = \mathrm{root}$ because of the special case in decanonize.) Let $V_{\mathsf{suff}} = \{q_1, \ldots, q_k\}$ be the nodes that are on the path from q to $p^{\mathsf{suff}}_{\mathsf{old}}$ ($V_{\mathsf{suff}} = \{\overline{\mathtt{a}}\}$ in our example). In the next call to $\mathrm{canonize}()$ in a suffix iteration, the nodes in V_{suff} have to be traversed

additionally. The purses that are expected to have a coin are assigned to the characters with the indices $I_{\mathsf{suff}} = \{i_j | i_j = n - |\alpha(t)| + \mathrm{depth}(q_j) + 1, q_j \in V_{\mathsf{suff}}\}$. We now show that the coins come from the reverse iteration.

In the reverse iteration, asp_{suffix} was changed by an assignment from app_{suffix}. Usually, app_{suffix} is decanonized in Step 5 of the reverse iteration. In the case, that the active suffix grows in a reverse iteration, the suffix location of app_{suffix} is represented by a suffix leaf (by Lemma 3.15). The node is found in constant time by Extension 1. Hence, there are coins that are not used in "prefix/decanonize" purses because $\mathrm{decanonize}()$ is not executed. (In the example, these are the coins that would be needed to decanonize over $\overline{\mathsf{a}}$). Let $p^{\mathsf{pref}}_{\mathsf{old}}$ be the base of $app_{\mathsf{suffix}}(t)$ and p^{pref}_{new} be the base of $app_{\mathsf{suffix}}(at)$. We know that $p^{\mathsf{pref}}_{new} = p^{\mathsf{suff}}_{new}$ and, by Corollary 3.5, we have $|\alpha(t)| + 1 = |\alpha(ta)| = |\hat{\alpha}(ta)| \leq |\hat{\alpha}(t)| + 1$, so $|\alpha(t)| \leq |\hat{\alpha}(t)|$, and $p^{\mathsf{pref}}_{\mathsf{old}}$ is a descendant or equals to $p^{\mathsf{suff}}_{\mathsf{old}}$. Therefore, if we had decanonized app_{suffix} (instead of assigning it directly to the suffix leaf), the nodes V_{pref} ($V_{\mathsf{pref}} = \{\overline{\mathsf{a}}\}$ in the example) that we would have met in $\mathrm{decanonize}()$ would be a superset of the nodes in V_{suff} (i.e., $V_{\mathsf{suff}} \subseteq V_{\mathsf{pref}}$). The string indices are $I_{\mathsf{pref}} = \{i_j | i_j = -m + |\hat{\alpha}(t)| - \mathrm{depth}(q_j) + 1, q_j \in V_{\mathsf{pref}}\}$. Obviously, $|I_{\mathsf{pref}}| \geq |I_{\mathsf{suff}}|$ and, because the coins were not needed for decanonizing app_{suffix}, we can take the money from the "prefix/decanonize" purses of the characters with indices I_{pref} and put one coin in the "suffix/canonize" purse of each character with an index in I_{suff}. (In the analysis of $\mathrm{decanonize}()$, we assumed that the coin from the characters with indices I_{pref} were used because we moved the offset index over them, but $\mathrm{decanonize}()$ was not executed.) As a result, Invariant I is kept and the additional work can be payed for. □

Further prefix iterations, which occur before $\mathrm{canonize}()$ is called in a suffix iteration, can lengthen the active suffix again, leading to the same money transfer. In iterations where the active suffix is unchanged, only new suffix nodes can be inserted.

Lemma 3.20.
Only constant work is needed to keep Invariant I when adding additional suffix nodes in prefix iterations.

Proof. Let p^{suff}_{new}, $p^{\mathsf{suff}}_{\mathsf{old}}$, q be the nodes of a previous prefix iteration $t = t_{-m} \dots t_n$ to $at = t_{-m-1} \dots t_n$ as described above in the proof of Lemma 3.19 (where the active suffix has grown and the active suffix has not been canonized yet). For each node r that is inserted between q and $p^{\mathsf{suff}}_{\mathsf{old}}$, an additional repetition of the while loop is needed in a successive call to $\mathrm{canonize}()$ (V_{suff} grows). The index of the character with the "suffix/canonize" purse to that a coin has to be added is $i_r = n - |\alpha(t)| + \mathrm{depth}(r) + 1$.

The new suffix node cannot lie between two different $p^{\mathsf{suff}}_{\mathsf{old}}$, q pairs. Suppose $p'^{\mathsf{suff}}_{\mathsf{old}}$ and q' are the nodes of the next case where the active suffix was lengthened with $p^{\mathsf{suff}}_{new} = p'^{\mathsf{suff}}_{\mathsf{old}}$. A node r between $p'^{\mathsf{suff}}_{\mathsf{old}}$,$q'$ and $p^{\mathsf{suff}}_{\mathsf{old}}$, q must have depth d with $d > \mathrm{depth}(q) = \mathrm{depth}(p^{\mathsf{suff}}_{new}) - 1 = \mathrm{depth}(p'^{\mathsf{suff}}_{\mathsf{old}}) - 1$ and $d < \mathrm{depth}(p'^{\mathsf{suff}}_{\mathsf{old}})$. This is not possible.

There is at most one suffix node inserted per reverse iteration, so paying the additional coin only adds constant extra cost. □

Note that suffix iterations cannot insert any such nodes because all nodes are inserted above or at the base of the asp_{suffix}.

Lemma 3.21.
Canonizing $asp_{\mathsf{suffix}}(ta)$ during prefix iterations keeps Invariant I.

Proof. Canonizing increases the offset index o towards the end of the string, so no positions without coins are introduced. □

As a result, Invariant I is always kept valid.

3.4.3.3 Changes to asp_{prefix} in a Reverse Iteration—Invariant II

Changes to asp_{prefix} concern Invariant II only. The following results can be applied analogously to app_{suffix} (by exchanging "suffix" and "prefix") and Invariant IV.

By Lemma 3.18, the invariant might only be destroyed if the offset index o of asp_{prefix} is increased (in the prefix view of the string).

Lemma 3.22.
Assigning $app_{\mathsf{prefix}}(at)$ to $asp_{\mathsf{prefix}}(at)$ in Step 6 adds amortized constant overhead to successive calls to $\mathrm{decanonize}(asp_{\mathsf{prefix}})$ and keeps Invariant II.

Proof. Let $t = t_{-m} \cdots t_n$. All prefix nodes that might have been added during Step 4 in this prefix iteration (if any) are added at a depth larger than the length of $\alpha(at)$. Assigning $app_{\mathsf{prefix}}(at)$ to $asp_{\mathsf{prefix}}(at)$ is equivalent to increasing the length of $asp_{\mathsf{prefix}}(t)$ by one. If $asp_{\mathsf{prefix}}(t)$ was canonical with base p_{old}, then the base p_{new} of the canonical reference pair $asp_{\mathsf{prefix}}(ta)$ is either the same as p_{old} or a child of p_{old}. In the latter case, $\mathrm{depth}(p_{\mathsf{new}}) = |\alpha(at)|$ because there cannot be a node q with $\mathrm{depth}(q) \leq |\alpha(t)|$ if $asp_{\mathsf{prefix}}(t)$ was canonical. We can put a coin in the "suffix/decanonize" purse of t_n to keep Invariant II. Because we need to put at most one additional coin in a "suffix/decanonize" per iteration, this adds at most constant overhead. □

We look at canonizing asp_{prefix} when the active suffix does not grow. We only need to canonize if new prefix nodes are inserted. The canonization can be seen as taking place each time a new node is inserted. For each new node, we check whether the node is the new canonical base for asp_{prefix}. The following lemma deals with all inserted nodes.

Lemma 3.23.
Only amortized constant work is needed to keep Invariant II when adding additional prefix nodes in prefix iterations.

Proof. Let o_{min} be the minimal value that the offset index ever had in all previous iterations. For a node q, let i_q be the index with which it would appear in a call to $\mathrm{decanonize}()$ (in a suffix iteration). We have $i_q = n - \mathrm{depth}(q) - 1$ (the transformed index for the prefix view is $-1 - i_q = \mathrm{depth}(q) - n$, whereas $n - \mathrm{depth}(q) - 1$ is the real index). Let r be the node with the largest string depth such that $-1 - i_r < o_{\mathsf{min}}$, and let p be the current base of asp_{prefix}.

A node q inserted below r does not destroy Invariant II because there is still the originally inserted coin on character t_{i_q}. We need to provide extra money for the nodes inserted above r to keep the invariant. Let q be inserted above r and below p with $-1 - i_q \geq o_{\mathsf{min}}$. We add a coin to the "suffix/decanonize" purse of character t_{i_q}

(prefix view of the string). If q is inserted above p so that asp_{prefix} is canonized to the new base $p' = q$, we add a coin to the "suffix/decanonize" purse of character t_{i_p}, and q becomes the new base. By the above accounting, we have now the situation that below o_{min} there is a coin in every character's purse, and there is a coin in each purse for a character corresponding to a node between o_{min} and o.

When $\mathrm{decanonize}()$ is called on asp_{prefix} and we move over a node q in the while-loop whilst o is still larger than o_{min}, the above payment guarantees that there is a coin in the "suffix/decanonize" purse of character t_o. We prove by induction.

Base case. We have not yet moved over a prefix edge. In this case, we meet exactly the same nodes for that we have added the coins for the corresponding characters.

Inductive case. We have moved over a prefix edge before. We have $i_q = n - \mathrm{depth}(q) - 1$ and q is a prefix node. Let q' be the suffix parent of q. By Lemma 3.2, there is a suffix parent and $\mathrm{depth}(q) = \mathrm{depth}(q') + 1$. By induction, there is a coin in the "suffix/decanonize" purse of character $i_{q'}$. The index is $i_{q'} = n' - \mathrm{depth}(q') - 1$, where n' was the rightmost string index in the step when the base was moved along the prefix edge the last time.

The base of asp_{prefix} is moved over a prefix edge only if a character is appended. Hence, we have $n' = n-1$ and $i_{q'} = n-1-\mathrm{depth}(q')-1 = n-1-(\mathrm{depth}(q)-1) - 1 = n - \mathrm{depth}(q) - 1$. Thus, there is a coin in the "suffix/decanonize" purse of character t_{i_q}.

For a string t of length n, we insert at most $2n$ prefix nodes in reverse iterations, thus we only pay an amortized constant amount of additional coins. □

Note again that suffix iterations cannot insert any such prefix nodes because all prefix nodes are inserted above or at the base of the asp_{prefix}.

As a result, Invariant II is always kept valid.

Chapter 4

Average-Case Analysis of Approximate Trie Search

In this chapter, we study the average-case behavior of searching in a dictionary (a set S) of strings allowing a certain number d of errors under a comparison-based string distance (see Section 2.1.3). In the notation of the classification scheme from Section 2.4, we look at indexing problems of the type $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{all}\rangle$ and $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{pref}\rangle$, where d is any comparison-based string distance, $n = |S|$ is the number of indexed strings, and $d = f(n)$ is the number of mismatches allowed, which we study as a function of n. Dictionary indexing is a well-known task, e.g., for looking up misspelled words. There exist very fast on-line search algorithms. For example, for comparison-based string distances, a search pattern u of length m can be interpreted as a deterministic finite-state automaton with $(m + 1)(d + 1)$ states. The automaton has a very regular structure, and it is possible to implement it by fast register operations. Starting the automaton on each string in S can—depending on the string distance and the number of allowed mismatches—thus be very fast. On the other hand, using a trie has the advantage that the prefixes of strings in S are combined. When we compare a prefix v of length k of the search pattern with a path spelling out the word w in the trie, we have effectively compared the prefix v with all strings in S having prefix w. In terms of the number of comparisons, this is always more efficient. In practice, traversing a trie has some extra overhead, e.g., in random-accesses to the memory, while comparing the pattern with a string is very simple. Thus, the latter approach can be implemented with a smaller constant in the asymptotic running time (and also with less space). We expect to see some trade-off or a threshold where one method outperforms the other.

To analyze the relative performance of the algorithms, we look at the average-case number of comparisons performed. We present a very exact analysis for that the constants in the asymptotics can be computed exactly for any concrete instance of the problem. A worst-case comparison cannot be used for a meaningful comparative analysis because it only deals with a small subset of possible inputs. An individual analysis of the algorithm implementations can contribute the constant factors involved in a single comparison. Thus, our analysis helps to decide which algorithm should be used for any concrete instance.

We discuss alternative algorithms in the introduction and a concrete alternative (for

Algorithm 4.1 LS(*set* S, *string* w, *integer* d)

Parameters: $S = \{t^1, \ldots, t^n\}$ is the collection of strings to be searched, w is the search pattern, and d the error bound.

```
1:  for j from 1 to n do
2:      i := 1
3:      c := 0
4:      l := min{|w|, |t^j|}
5:      while c ≤ d do
6:          while i ≤ l and d(w_i, t^j_i) = 0 do
7:              i := i + 1
8:          c := c + 1
9:          i := i + 1
10:     if i = |w| then
11:         report a match for t^j
```

a constant number of errors) in Chapter 5.

4.1 Problem Statement

Let $S = \{t^1, \ldots, t^n\} \subset \Sigma^*$ be a collection of strings to be indexed with $n = |S|$. In the context of tries or direct comparison, checking whether a prefix $t^j{}_{-,i}$ of a string $t^j \in S$ is the complete string is an easy task. The algorithms considered in this chapter can provide solutions to $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{all}\rangle$ and $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{pref}\rangle$ with only a minor modification of the reporting function. We therefore consider only the latter problem and report a document if a prefix is matched.

We assume that all strings are generated independently at random by a memoryless source with uniform probabilities, i.e., the probability that a is the i-th character of the j-th string is given by $\Pr\{t^j_i = a\} = \frac{1}{|\Sigma|}$. We will be using the size of the alphabet $|\Sigma|$ a lot, and therefore abbreviate it by $\sigma = |\Sigma|$. For the average-case analysis, it is convenient to assume that we deal with strings of infinite length. Having strings of finite length in practice has only a negligible impact on the validity of our analysis: If the end of a string is reached, no more comparisons take place, thus the theoretical model gives an upper bound. Furthermore, if the strings are sufficiently large, the probability of ever reaching the end is very small.

We assume that the search pattern w is also generated independently at random by a memoryless source with uniform probabilities. Comparison-based string distances are completely defined at the character level. For a given distance d, the parameters relevant to our analysis are thus the probabilities of a match or a mismatch given by

$$q = \frac{1}{\sigma^2} \sum_{a \in \Sigma} \sum_{b \in \Sigma} \mathrm{d}\,(a, b) \quad \text{and} \quad p = \frac{1}{\sigma^2} \sum_{a \in \Sigma} \sum_{b \in \Sigma} (1 - \mathrm{d}\,(a, b)) = 1 - q \ . \tag{4.1}$$

For example, for Hamming distance, we have a mismatch probability of $q = 1 - \frac{1}{\sigma}$ and a match probability of $p = \frac{1}{\sigma}$.

Pseudo code for both algorithms is given in Figures 4.1 and 4.2. The LS algorithm (for "Linear Search") just compares the search pattern with each string. The TS algo-

---

Algorithm 4.2 TS(*trie* $\mathcal{T}$, *node* v, *string* w, *integer* i, *integer* c)

Parameters: $\mathcal{T}$ is a trie for the collection of strings $S = \{t^1, \ldots, t^n\}$, v is a node of $\mathcal{T}$ (in the first call v is the root), w is search pattern, i is the current position in w (zero in the first call), and c is the maximal number of mismatches allowed (in the first call we have $c = d$).

```
1:  if c ≥ 0 then
2:    if v is a leaf then
3:      report match for t^value(v)
4:    else if i > |w| then
5:      for all leaves u in the subtree T_v rooted at v do
6:        report match for t^value(u)
7:    else
8:      for each child u of v do
9:        let a be the edge label of the edge (u, v)
10:       if d(w_i, a) = 0 then
11:         TS(T, u, w, i + 1, c)
12:       else
13:         TS(T, u, w, i + 1, c − 1)
```

---

rithm (for "Trie Search") uses a previously constructed trie to reduce the number of comparisons. We store $\mathrm{value}(v) = i$ at leaf v if the path to v spells out the string t^i. On a random set S, let L_n^d be the number of comparisons made by the LS algorithm and T_n^d be the number of comparisons made by the TS algorithm using a trie as an index. We are interested in the average-case; therefore, we compute bounds on the expected values $\mathbb{E}[L_n^d]$ and $\mathbb{E}[T_n^d]$. In the analysis, we assume that q and p are given parameters and look at the behavior for different d. What is the threshold for d (seen as a function $d = f(n)$) up to where the average $\mathbb{E}[T_n^d]$ is asymptotically smaller than the average $\mathbb{E}[L_n^d]$? What is the effect of different mismatch probabilities? The answers to these questions give us the clues needed to choose the more efficient of both methods in various situations.

4.2 Average-Case Analysis of the LS Algorithm

Computing the expected behavior of L_n^d involves only some basic algebra with generating functions. Because of our assumptions, we can also derive convergence probabilities for the behavior of L_n^d. It indeed comes as no surprise that L_n^d is so well behaved because we are in essence dealing with a huge amount of independent Bernoulli trials so that the Law of Large Numbers results in a very predictable (in the sense of likely) behavior.

If we only have one string, the probability that Algorithm 4.1 makes k comparisons is given by

$$\Pr\left\{L_1^d = k\right\} = \binom{k-1}{d} q^{d+1} p^{k-d-1} \quad . \tag{4.2}$$

For n strings, we have to sum over all possibilities to distribute the k comparisons over

the strings. Because $\binom{k}{l} = 0$ for $k < l$, we get

$$\Pr\left\{L_n^d = k\right\} = \sum_{i_1+\cdots+i_n=k} \prod_{j=1}^{n} \binom{i_j - 1}{d} q^{d+1} p^{i_j-d-1} \quad . \tag{4.3}$$

We use this equation to derive the probability generating function $g_{L_n^d}(z)$:

$$\begin{aligned} g_{L_n^d}(z) = \mathbb{E}\left[z^{L_n^d}\right] &= \sum_{k=0}^{\infty} \Pr\left\{L_n^d = k\right\} z^k \\ &= \sum_{k=0}^{\infty} \sum_{i_1+\ldots+i_n=k} \prod_{j=1}^{n} \binom{i_j - 1}{d} q^{d+1} p^{i_j-d-1} z^k \\ &= \left(\sum_{k=0}^{\infty} \binom{k-1}{d} q^{d+1} p^{k-d-1} z^k\right)^n = \left(\frac{zq}{1-zp}\right)^{n(d+1)} \quad . \end{aligned} \tag{4.4}$$

Thus, we find the expectation and variance of L_n^d to be

$$\mathbb{E}\left[L_n^d\right] = \frac{d+1}{q} n \quad \text{and} \quad \mathbb{V}\left[L_n^d\right] = \frac{(d+1)p}{q^2} n \quad . \tag{4.5}$$

As already mentioned, the stochastic process is very stable. We can use Chebyshev's inequality to derive convergence in probability of L_n^d to its mean.

$$\begin{aligned} &\Pr\left\{\left|\frac{L_n^d}{n(d+1)} - \frac{1}{q}\right| > \epsilon\right\} \\ &\qquad = \Pr\left\{\left|L_n^d - \frac{n(d+1)}{q}\right| > \epsilon n(d+1)\right\} \\ &\qquad\qquad < \frac{p}{q^2 \epsilon^2 n(d+1)} \quad . \end{aligned} \tag{4.6}$$

Hence, we have proven convergence in probability for

$$\lim_{n\to\infty} \frac{L_n^d}{n(d+1)} = \frac{1}{q} \qquad \text{(pr)} \quad .$$

Note that, if $d = f(n)$ is a function of n, then we already have almost sure convergence if $f(n) = \omega(\log^{1+\delta} n)$ for some $\delta > 0$ by a simply application of the Borel-Cantelli Lemma (see, e.g., [Kal02]) and the fact that

$$\sum_{n=0}^{\infty} \Pr\left\{\left|\frac{L_n^d}{n(d+1)} - \frac{1}{q}\right| > \epsilon\right\} < \sum_{n=0}^{\infty} \frac{p}{q^2 \epsilon^2 n \log^{1+\delta} n} < \infty \tag{4.7}$$

because $\sum_{n\geq 0} n^{-1} \log^{-(1+\epsilon)} n$ is convergent. Using a method of Kesten and Kingman (see [Kin73] or [Szp00]), this can be extended to almost sure convergence: First, note that $L_n^d < L_{n+1}^d$, so L_n^d is non-decreasing. For any positive constant s let $r = r(n)$ be the largest integer with $r^s \leq n$, then $L_{r^s}^d \leq L_n^d \leq L_{(r+1)^s}^d$, and thus

$$\begin{aligned} \limsup_{n\to\infty} \frac{L_n^d}{n(d+1)} &\leq \limsup_{r\to\infty} \frac{L_{r^s}^d}{(r+1)^s(d+1)} \\ &= \limsup_{r\to\infty} \frac{L_{r^s}^d}{r^s(d+1)} \frac{r^s}{(r+1)^s} \leq \limsup_{r\to\infty} \frac{L_{r^s}^d}{r^s(d+1)} \end{aligned} \tag{4.8}$$

and equally

$$\liminf_{n\to\infty} \frac{L_n^d}{n(d+1)} \geq \liminf_{r\to\infty} \frac{L_{(r+1)^s}^d}{r^s(d+1)}$$
$$= \liminf_{r\to\infty} \frac{L_{(r+1)^s}^d}{(r+1)^s(d+1)} \frac{(r+1)^s}{r^s} \geq \liminf_{r\to\infty} \frac{L_{(r+1)^s}^d}{(r+1)^s(d+1)} \quad . \tag{4.9}$$

By the Borel-Cantelli Lemma, $\frac{L_{r^s}^d}{r^s(d+1)}$ converges to $\frac{1}{q}$ almost surely, because for all $s \geq 1 + \epsilon$ we have

$$\sum_{r=0}^{\infty} \Pr\left\{\left|\frac{L_{r^s}^d}{r^s(d+1)} - \frac{1}{q}\right| \geq \epsilon\right\} \leq \sum_{r=0}^{\infty} \frac{p}{q^2\epsilon^2(d+1)r^s} = \frac{p}{q^2\epsilon^2(d+1)}\zeta(s) < \infty \tag{4.10}$$

(where $\zeta(s)$ is the Riemann zeta function), and thus

$$\lim_{r\to\infty} \frac{L_{r^s}^d}{r^s(d+1)} = \frac{1}{q} \qquad \text{(a.s.)} \quad . \tag{4.11}$$

As a result, we find that

$$\lim_{n\to\infty} \frac{L_n^d}{n(d+1)} = \frac{1}{q} \qquad \text{(a.s.)} \quad .$$

Note that, interpreting $d = f(n)$ as a function of n, this also holds for $f(n) = \Omega(1)$.

4.3 Average-Case Analysis of the TS Algorithm

The behavior of T_n^d is somewhat harder to analyze. We pursue the following plan: We start by setting up a recursive equation for the expected number of comparisons. Using some algebra on generating functions, we arrive at an exact formula. Unfortunately, the formula requires a summation over n terms of exponential size and with alternating signs. Therefore, we apply some more advanced methods from complex function analysis to derive an asymptotic bound. The number of terms in the exact formula for the bound is polynomial in d (where one term is itself an infinite sum), but it gives us precise bounds using the highest order terms and will enable us to determine parameter ranges for that the TS algorithm is the better choice.

Instead of counting the number of comparisons, it is easier to examine the number of nodes visited in the trie. Observe that the number of nodes visited is by one larger than the number of character comparisons. We denote the number of nodes visited by the process in a trie for n strings allowing d mismatches by $T_n'^d$.

4.3.1 An Exact Formula

We derive a recursive equation for the expected number of nodes visited. Let $g_n^d = \mathbb{E}[T_n'^d] = \mathbb{E}[T_n^d] + 1$. At any node u, we have at most σ children $v_1, \ldots, v_\sigma$. Each child is the root of a subtree having i_j children. Let d be the number of allowed mismatches at node u. When the algorithm examines node v_j, the number of allowed

mismatches is either d with probability p, or it is $d-1$ with probability q. Thus, we get the desired recursive equation by summing over all possibilities to partition the n strings in the subtree of u into the children's subtrees. The probability for a partition into subtrees of sizes $i_1, i_2, \ldots, i_\sigma$ is

$$\Pr\{i_1 + i_2 + \cdots + i_\sigma = n\} = \frac{n!}{i_1! \cdots i_\sigma!}\frac{1}{\sigma^n} = \binom{n}{i_1, \ldots, i_\sigma}\sigma^{-n} \quad . \tag{4.12}$$

The following recursive equation is valid for $n > 0$ and $d \geq 0$:

$$g_n^d = 1 + \sum_{i_1+\cdots+i_\sigma=n} \binom{n}{i_1, \ldots, i_\sigma}\sigma^{-n}\left(q\sum_{j=1}^{\sigma} g_{i_j}^{d-1} + p\sum_{j=1}^{\sigma} g_{i_j}^{d}\right) \quad . \tag{4.13}$$

If there is no string represented in a subtree, we make no comparisons. Hence, we have $g_0^d = 0$ for all d. If we reach a node with $d = -1$, we had a mismatch in the last character comparison and do not continue (line one of Algorithm 4.2); therefore, we have $g_n^{-1} = 1$. We can easily check that, if there is only one string left, by equation (4.13), we make $g_1^d = 1 + \frac{d+1}{q}$ comparisons. This agrees with $1 + \mathbb{E}[L_1^d]$.

We derive the exponential generating function of g_n^d by summing both sides of equation (4.13) for $n \geq 0$. We account the case $n = 0$, which in equation (4.13) wrongly gives $g_0^d = 1$, by subtracting one on the right hand side. Thus, the equation for $G_d(z) = \sum_{n\geq 0} g_n^d \frac{z^n}{n!}$ is

$$G_d(z) = e^z + q\sum_{j=1}^{\sigma} G_{d-1}\left(\frac{z}{\sigma}\right)e^{\left(1-\frac{1}{\sigma}\right)z} + p\sum_{j=1}^{\sigma} G_d\left(\frac{z}{\sigma}\right)e^{\left(1-\frac{1}{\sigma}\right)z} - 1 \quad . \tag{4.14}$$

In the above, we have already simplified the σ-fold

$$\begin{aligned}\sum_{n\geq 0}\sum_{i_1+\cdots+i_\sigma=n} \binom{n}{i_1, \ldots, i_\sigma}\sigma^{-n} G_{i_j}^d \frac{z^n}{n!} \\ = \sum_{n\geq 0}\sum_{i_1+\cdots+i_\sigma=n} \frac{z^{i_1}}{\sigma^{i_1} i_1!}\cdots\frac{G_{i_j}^d z^{i_j}}{\sigma^{i_j} i_j!}\cdots\frac{z^{i_\sigma}}{\sigma^{i_\sigma} i_\sigma!} \\ = G_d\left(\frac{z}{\sigma}\right)e^{\left(1-\frac{1}{\sigma}\right)z} \quad . \end{aligned}\tag{4.15}$$

We multiply both sides by e^{-z} (which corresponds to applying some kind of binomial inversion) and define $\tilde{G}_d(z) = G_d(z)e^{-z}$. We now have

$$\tilde{G}_d(z) = 1 - e^{-z} + \sigma q \tilde{G}_{d-1}\left(\frac{z}{\sigma}\right) + \sigma p \tilde{G}_d\left(\frac{z}{\sigma}\right). \tag{4.16}$$

Let $\tilde{g}_n^d$ be the coefficients of $\tilde{G}_d(z)$. Then we have

$$\tilde{g}_n^d = (-1)^n \sum_{k=0}^{n}\binom{n}{k}(-1)^k g_k^d \quad \text{and} \quad g_n^d = \sum_{k=0}^{n}\binom{n}{k}\tilde{g}_k^d \quad . \tag{4.17}$$

For $\tilde{g}_n^d$, we get the boundary conditions $\tilde{g}_1^d = 1 + \frac{(d+1)}{q}$, $\tilde{g}_0^d = 0$, $\tilde{g}_n^{-1} = (-1)^{n-1}$ for $n > 0$, and $\tilde{g}_0^{-1} = 0$. Comparing coefficients in equation (4.16), we find for $n > 1$ that

$$\tilde{g}_n^d = \frac{(-1)^{n-1} + \tilde{g}_n^{d-1}\sigma^{1-n}q}{1-\sigma^{1-n}p} \quad , \tag{4.18}$$

which by iteration (on d) leads to

$$\tilde{g}_n^d = \frac{(-1)^n}{1-\sigma^{1-n}}\left(\sigma^{1-n}\left(\frac{\sigma^{1-n}q}{1-\sigma^{1-n}p}\right)^{d+1} - 1\right) \quad . \tag{4.19}$$

Finally, combining equations (4.17) and (4.18) proves the following lemma.

Lemma 4.1 (Exact expected number of nodes visited by Algorithm 4.2).
The expected number of nodes visited by Algorithm 4.2 on a random trie of n strings with a random pattern is

$$g_n^d = n\left(1 + \frac{d+1}{q}\right) - \sum_{k=2}^{n}\binom{n}{k}\frac{(-1)^k}{1-\sigma^{1-k}} + \sum_{k=2}^{n}\binom{n}{k}\frac{(-1)^k}{\sigma^{k-1}-1}\left(\frac{q\sigma^{1-k}}{1-p\sigma^{1-k}}\right)^{d+1} \quad . \tag{4.20}$$

We can divide (4.20) into three parts. Each part has its own interpretation and will be handled separately:

$$g_n^d = \underbrace{1 + \frac{d+1}{q}n}_{1+\mathbb{E}\left[L_n^d\right]} + \underbrace{(n-1) - \sum_{k=2}^{n}\binom{n}{k}\frac{(-1)^k}{1-\sigma^{1-k}}}_{-\mathcal{A}_n} + \underbrace{\sum_{k=2}^{n}\binom{n}{k}\frac{(-1)^k}{\sigma^{k-1}-1}\left(\frac{q\sigma^{1-k}}{1-p\sigma^{1-k}}\right)^{d+1}}_{\mathcal{S}_n^d} \quad . \tag{4.21}$$

This finishes the first part of our derivation. We handle the two sums in equation (4.20) separately starting with an excursion on $\mathcal{A}_n$. We use Rice's integrals to tackle both sums. Before embarking on these tasks, we describe a general scheme how to translate these sums into integrals involving only the Gamma function (instead of the Beta function). This makes the problem easier. Along the way, we will also see how the sum can be described by a simple line integral.

4.3.2 Approximation of Integrals with the Beta Function

Using Theorem 2.2 on equation (4.21) yields a path integral involving the Beta function. The evaluation of the residues involving the Beta function is tricky. Therefore, we approximate the Beta function using the asymptotic expansion given by equation (2.32) introduced in Section 2.5. Using this approximation was already suggested by Szpankowski [Szp88a]—although without commenting on validity conditions. In the following, we will lay a rigorous basis for it. The main idea is to replace the Beta function in an integral of the form

$$\frac{1}{2\pi\imath}\oint_{\mathcal{C}} f(n,z)\mathrm{B}(n+1,-z)dz \quad , \tag{4.22}$$

where $\mathcal{C}$ is a path encircling some poles of $\mathrm{B}(n+1,-z)$ at $\{a,\ldots,b\}$, by a polynomial in z and the Gamma function. This can be done using the asymptotic expansion given in equation (2.32). We assume in the following that $f(n,z)$ is a function of polynomial growth in n and constant growth in z, i.e., $|f(n,z)| = O(n^r)$ for a constant r, then $|f(n,z)| = O(1)$ for $z \to \infty$. The growth of $\mathrm{B}(n+1,-z) = \frac{(-1)^n n!}{z(z-1)\cdots(z-n)}$ is $O(z^{-n-1})$ as $z \to \infty$ (in relation to z, n can be considered as a constant). The first step is to bound the value of the integral along a circle with large radius.

Lemma 4.2 (The Beta function integral along arcs with large radius).
Let $f(n,z)$ be a function such that $|f(n,z)| = O(n^r)$ for a constant r. Let $\mathcal{C}$ be a path corresponding to a part of a circle in the complex plane with radius m, i.e., $\mathcal{C} = \{p \mid p = m{\cdot}e^{\imath t} \text{ for } a \le t \le b\}$ where $a,b \in [0,\pi)$, and assume that $\mathcal{C}$ is such that it avoids any singularities. Then

$$\left|\frac{1}{2\pi\imath}\int_{\mathcal{C}} f(n,z)B(n+1,-z)dz\right| \xrightarrow[m\to\infty]{} 0 \ . \tag{4.23}$$

Proof. The proof follows easily from

$$\begin{aligned}
&\left|\frac{1}{2\pi\imath}\int_{\mathcal{C}} f(n,z)\mathrm{B}(n+1,-z)dz\right| \\
&\qquad = \left|\frac{1}{2\pi}\int_a^b f(n,z)\frac{(-1)^n n!}{me^{\imath t}(me^{\imath t}-1)\cdots(me^{\imath t}-n)}dt\right| \\
&\qquad \le \frac{1}{2\pi}\int_a^b |f(n,z)|\left|\frac{(-1)^n n!}{me^{\imath t}(me^{\imath t}-1)\cdots(me^{\imath t}-n)}\right|dt \\
&\qquad\qquad \le \frac{c(b-a)}{2\pi}n^r\frac{n!}{(m-n)^{n+1}} \xrightarrow[m\to\infty]{} 0 \ .
\end{aligned} \tag{4.24}$$

□

A simple consequence of the above lemma for equation (4.22) is the following: Assume that the path $\mathcal{C}$ encircles the poles at $\{a,\ldots,b\}$ and that there are no other singularities in the half-plane $\Re(z) \ge a-\epsilon$ for some $\epsilon > 0$, then we can extend $\mathcal{C}$ to a vertical line at $\Re(z) = a-\epsilon$ and a half-circle with radius $m \to \infty$ such that

$$\frac{1}{2\pi\imath}\oint_{\mathcal{C}} f(n,z)\mathrm{B}(n+1,-z)dz = \frac{1}{2\pi\imath}\int_{a-\epsilon+\imath\infty}^{a-\epsilon-\imath\infty} f(n,z)\mathrm{B}(n+1,-z)dz \ . \tag{4.25}$$

The simple line integral allows us to apply the approximation. We can then return to a path integral later.

Lemma 4.3 (Asymptotic approximation of a Beta function integral).
For real constant $x \notin \{0,-1,-2,\ldots\}$, we have

$$\int_{-\infty}^{\infty} |B(n,x+\imath y)|dy = O\left(n^{-x}\right) \ . \tag{4.26}$$

Proof. The plan of the proof is as follows. We rewrite the Beta function as ratio of Gamma functions. Then we use Stirling's formula to approximate these. The resulting

integral has four different factors in the integrand, for which we make a case distinction, depending on the range of y with respect to n.

We start simplifying by applying Stirling's formula:

$$\int_{-\infty}^{\infty} |\mathrm{B}(n, x+\imath y)| dy = 2\int_{0}^{\infty} |\mathrm{B}(n, x+\imath y)| dy$$
$$= 2\int_{0}^{\infty} \frac{\Gamma(n)|\Gamma(x+\imath y)|}{|\Gamma(n+x+\imath y)|} dy = 2\int_{0}^{\infty} \sqrt{2\pi}\left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}}$$
$$\cdot \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^{x} \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \left(1+O\left(n^{-1}\right)\right) \quad , \qquad (4.27)$$

where $0 \le \operatorname{sgn}(y)\phi(n,x,y) = \operatorname{sgn}(y)(\arg(x+\imath y) - \arg(x+n+\imath y)) < \pi$. The latter follows from the fact that, for $y \neq 0$, we have

$$|\cos(\phi(n,x,y))| = \left|\frac{nx+x^2+y^2}{\sqrt{x^2+y^2}\sqrt{n^2+2nx+x^2+y^2}}\right| < 1 \ . \qquad (4.28)$$

Because $\arg(z) < 0$ for $\Im(z) = y < 0$, we analyze $\phi(n,x,y)$ as a function $\phi(y)$ of y for $y \ge 0$. Note that $\cos(\phi(y))$ grows inverse to $\phi(y)$ on $[0,\pi]$. The derivative of $\cos(\phi(y))$ is

$$\frac{d}{dy}\cos(\phi(n,x,y)) = \frac{yn^2(-nx-x^2+y^2)}{(x^2+y^2)^{\frac{3}{2}}(n^2+2nx+x^2+y^2)^{\frac{3}{2}}} \quad , \qquad (4.29)$$

which is zero for $y = 0$ and for $y = \sqrt{nx+x^2}$. The latter is impossible for $x < 0$ and $|x| < n$. For $x < 0$ and $|x| < n$, we have $\cos(\phi(y)) = -1$ and $\phi(y) = \frac{\pi}{2}$ for $y = 0$. Otherwise we have $\cos(\phi(y)) = 1$ and $\phi(y) = 0$ at $y = 0$. As $y \to \infty$ we have $\cos(\phi(y)) \to 1$ and similarly $\phi(y) \to 0$. We consider x a constant with respect to n. Thus, for $x < 0$, $\phi(y)$ decreases from π to zero and, for $x > 0$, $\phi(y)$ grows from zero to a maximum at $y = \sqrt{nx+x^2}$ and then decreases back to zero. As a result, the term $e^{-y\phi(n,x,y)}$ is monotonically decreasing in y.

The term $\frac{n}{|n+x+\imath y|}$ is monotonically decreasing in y and tends to zero. The term $\frac{|x+\imath y|}{|n+x+\imath y|}$ is monotonically increasing in y and tends to one.

We make a case distinction for the integration interval. For $x > 0$, we find that

$$\int_{0}^{x} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^{x} \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy$$
$$\le \left(\frac{n}{n+x}\right)^{n-\frac{1}{2}} \left(\sqrt{\frac{x^2+x^2}{n^2+2nx+2x^2}}\right)^{x} \frac{1}{\sqrt{x}} \int_{0}^{x} e^{0} dy$$
$$\le \left(\sqrt{2}x\right)^{x} n^{-x}\sqrt{x} = O\left(n^{-x}\right) \ . \qquad (4.30)$$

Secondly, for n large enough, we have

$$\begin{aligned}
&\int_x^n \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^x \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \\
&\quad\leq \left(\sqrt{\frac{n^2}{(n+x)^2+x^2}}\right)^{n-\frac{1}{2}} \frac{1}{\sqrt{2x}} \int_x^n \left(\sqrt{\frac{x^2+y^2}{n^2}}\right)^x e^{-y\phi(n,x,y)} dy \\
&\quad\leq \frac{n^{-x}}{\sqrt{2x}} \int_x^n \left(\sqrt{2}y\right)^x e^{-y\frac{\pi}{5}} dy \\
&\quad\leq \frac{n^{-x}}{\sqrt{2x}} \frac{5}{\pi} \left(\sqrt{2}\frac{5}{\pi}\right)^x \Gamma(x+1) = O\left(n^{-x}\right) \ , \qquad (4.31)
\end{aligned}$$

because $\phi(n,x,x) \xrightarrow[n\to\infty]{} \arccos\left(\frac{1}{\sqrt{2}}\right) = \frac{\pi}{4}$ and $\phi(n,x,n) \xrightarrow[n\to\infty]{} \arccos\left(\frac{1}{\sqrt{2}}\right) = \frac{\pi}{4}$, so we have $\phi(n,x,y) > \frac{\pi}{5}$ for some n large enough. The third range is

$$\begin{aligned}
&\int_n^{n^2} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^x \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \\
&\quad\leq \left(\sqrt{\frac{n^2}{(n+x)^2+n^2}}\right)^{n-\frac{1}{2}} \left(\sqrt{\frac{x^2+n^4}{(n+x)^2+n^4}}\right)^x (x^2+n^2)^{-\frac{1}{4}} n^2 \\
&\quad= O\left(2^{-\frac{n}{2}} n^2\right) \ . \qquad (4.32)
\end{aligned}$$

Finally, for the last part we have

$$\begin{aligned}
&\int_{n^2}^{\infty} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^x \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \\
&\quad\leq \int_{n^2}^{\infty} \left(\sqrt{\frac{n^2}{(n+x)^2+yn^2}}\right)^{n-\frac{1}{2}} \frac{1}{\sqrt{y}} dy \\
&\quad\leq \int_{n^2}^{\infty} y^{-\frac{n}{2}-\frac{1}{4}} dy = O\left(n^{-n-\frac{1}{2}}\right) \ . \qquad (4.33)
\end{aligned}$$

Thus, the whole integral is $O(n^{-x})$.

For $x < 0$ ($-x \notin \mathbb{N}$), the derivation is almost the same. We start with a slightly larger range:

$$\begin{aligned}
&\int_0^n \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^x \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \\
&\quad\leq \sqrt{\frac{n}{n+x}} \left(\frac{n}{n+x}\right)^n \int_0^n \left(\frac{|x+\imath y|}{|n+x+\imath y|}\right)^x \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy \\
&\quad\leq 4e^{-x} x^x (2n)^{-x} \frac{1}{\sqrt{x}} \int_0^n e^{-y\frac{\pi}{5}} dy \\
&\quad= n^{-x} \frac{4(2e)^{-x} x^x}{\sqrt{x}} \frac{-5}{\pi} (e^{-\frac{1}{5}n\pi} - 1) = O\left(n^{-x}\right) \ , \qquad (4.34)
\end{aligned}$$

because $\left(\frac{n}{n+x}\right)^n < 2e^{-x}$, $\sqrt{\frac{n}{n+x}} < 2$, and $\phi(n,x,y) > \frac{\pi}{5}$ for some n large enough (with $\phi(n,x,n) \xrightarrow[n\to\infty]{} = \frac{\pi}{4}$ and $\phi(n,x,0) = \pi$). The second part is

$$\int_n^{n^2} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|n+x+\imath y|}{|x+\imath y|}\right)^{-x} \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy$$
$$\leq \left(\sqrt{\frac{n^2}{(n+x)^2+n^2}}\right)^{n-\frac{1}{2}} \left(\sqrt{\frac{(n+x)^2+n^2}{x^2+n^2}}\right)^{-x} (x^2+n^2)^{-\frac{1}{4}} n^2$$
$$\leq \left(\sqrt{\frac{n^2}{\frac{3}{2}n^2}}\right)^{n-\frac{1}{2}} 2^{-\frac{x}{2}} n^2 = O\left(\left(\frac{3}{2}\right)^{-\frac{n}{2}} n^2\right) . \quad (4.35)$$

The final part is

$$\int_{n^2}^{\infty} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|n+x+\imath y|}{|x+\imath y|}\right)^{-x} \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy$$
$$\leq \int_{n^2}^{\infty} \left(\frac{n}{\sqrt{(n+x)^2+y^2}}\right)^{n-\frac{1}{2}} \left(\sqrt{\frac{2y^2}{y^2}}\right)^{-x} \frac{1}{\sqrt{y}} e^{-y\phi(n,x,y)} dy$$
$$\leq 2^{-\frac{x}{2}} \int_{n^2}^{\infty} y^{-\frac{n}{2}-\frac{1}{4}} dy$$
$$= O\left(n^{-n-\frac{1}{2}}\right) . \quad (4.36)$$

This finishes the proof. □

The above proof relies on the fact that the Beta function becomes very small very quickly along the imaginary axis. The next lemma details this.

Lemma 4.4 (Tail of a Beta function integral).
For $x < 0$ and any strictly positive function $f(n) \in \omega(1)$, we have

$$\int_{f(n)\ln n}^{\infty} |B(n, x+\imath y)| dy = O\left(n^{-f(n)\left(\frac{\pi}{4}-\epsilon\right)-x}\right) , \quad (4.37)$$

for some real constant $\epsilon > 0$.

Proof. We use the same derivations as in Lemma 4.3 together with the asymptotic behavior $\phi(n,x,n) \xrightarrow[n\to\infty]{} = \frac{\pi}{4}$. Thus, we find that

$$\int_{f(n)\ln n}^{n} \left(\frac{n}{|n+x+\imath y|}\right)^{n-\frac{1}{2}} \left(\frac{|n+x+\imath y|}{|x+\imath y|}\right)^{-x} \frac{1}{\sqrt{|x+\imath y|}} e^{-y\phi(n,x,y)} dy$$
$$\leq e^{-x} \left(\frac{f(n)\ln n}{\sqrt{2}n}\right)^x (\ln n)^{-\frac{1+\epsilon}{2}} \int_{f(n)\ln n}^{n} e^{-y\left(\frac{\pi}{4}-\epsilon\right)} dy$$
$$\leq e^{-x} \left(\frac{f(n)\ln n}{\sqrt{2}n}\right)^x (f(n)\ln n)^{-\frac{1}{2}} \frac{1}{\left(\frac{\pi}{4}-\epsilon\right)} e^{-f(n)\ln n\left(\frac{\pi}{4}-\epsilon\right)}$$
$$= O\left(n^{-f(n)\left(\frac{\pi}{4}-\epsilon\right)-x}\right) . \quad (4.38)$$

□

Lemma 4.5 (Approximation of Beta function integrals).
Let $f(n,z)$ be a function such that $|f(n,z)| = O(n^r)$ for a constant r (i.e., $f(n,z)$ is of constant growth with respect to z). Let $z = x + \imath y$. For $x < 0$, we can approximate for some constant c and fixed n

$$\int_{x-\imath\infty}^{x+\imath\infty} f(n,z)B(n,z)dz$$
$$= \sum_{k=0}^{N-1} \int_{x-\imath\infty}^{x+\imath\infty} f(n,z)\frac{(-1)^k B_k^{(1-z)}(1)}{k!}\Gamma(z+k)n^{-k-z}dz$$
$$+ c\int_{x-\imath\infty}^{x+\imath\infty} f(n,z)n^{-N-z}\Gamma(z)dz + o(1) \quad . \quad (4.39)$$

Proof. The proof relies on the standard expansion of the ratio of two Gamma functions ([TE51, AS65], see equation (2.32) in Section 2.5). We use the fact that the expansion is uniform [Fie70] and that the tails of the integrands are exponentially small (Lemma 4.4).

By equation (2.32), we can approximate $\mathrm{B}(n, x+\imath y) \le g_N(n, x+\imath y)$ on the interval $0 \le y \le n^{\frac{1}{3}}$ by

$$g_N(n, x+\imath y) = \sum_{k=0}^{N-1} \frac{(-1)^k B_k^{(1-x-\imath y)}(1)}{k!}\Gamma(x+\imath y+k)n^{-k-x-\imath y}$$
$$+\ cn^{-N-x-\imath y}|y|^{2N}\Gamma(x+\imath y) \quad . \quad (4.40)$$

For $y > n^{\frac{1}{3}}$, we use Lemma 4.4 and get

$$\int_{n^{\frac{1}{3}}}^{\infty} |f(n,z)\mathrm{B}(n,x+\imath y)|dy = O\left(n^r\right)\int_{n^{\frac{1}{3}}}^{\infty} |\mathrm{B}(n,x+\imath y)|dy$$
$$= O\left(e^{-n^{\frac{1}{3}}\left(\frac{\pi}{4}-\epsilon\right)}n^{-x+r}\right) \quad . \quad (4.41)$$

The terms of $f(n,z)g_N(n,x+\imath y)$ are of the type $n^{r-k-x-\imath y}\Gamma(x+\imath y+k)(x+\imath y)^l$ (for some $l \le k$). Recall that l, k, x have to be considered as constant and that $x < 0$. Integrating over a term like this yields

$$\int_{n^{\frac{1}{3}}}^{\infty} \left|n^{r-k-x-\imath y}\Gamma(x+\imath y+k)(x+\imath y)^l\right|dy$$
$$\le \sqrt{2\pi}n^{r-k-x}x^l \int_{n^{\frac{1}{3}}}^{\infty} y^l|x+\imath y+k|^{x+k-\frac{1}{2}}e^{-y\arg(x+k+\imath y)-x-k+\frac{x+k}{12((x+k)^2+y^2)}}dy$$
$$\le \sqrt{2\pi}e^{-x-k}n^{r-k-x}x^l|x+k|^{x+k}\int_{n^{\frac{1}{3}}}^{\infty} y^{x+k+l}e^{-y\arg(x+k+\imath y)}dy$$
$$= O\left(n^{r-k-x}\right)\int_{n^{\frac{1}{3}}}^{\infty} y^{2k}e^{-y\frac{\pi}{3}}dy = O\left(n^{r-k-x}\right)\int_{\frac{\pi}{3}n^{\frac{1}{3}}}^{\infty} y^{2k}e^{-y}dy$$
$$= O\left(n^{r-k-x+\frac{2}{3}k}e^{-\frac{\pi}{3}n^{\frac{1}{3}}}\right) \quad , \quad (4.42)$$

because we have $\arg(x + k + \imath y) \geq \frac{\pi}{3}\mathrm{sgn}(y)$, and

$$\int_{\frac{\pi}{3}n^{\frac{1}{3}}}^{\infty} u^{2k} e^{-u} du \leq 2\Big(\frac{\pi}{3} n^{\frac{1}{3}}\Big)^{2k} e^{-\frac{\pi}{3}n^{\frac{1}{3}}} \tag{4.43}$$

by iterated integration (for n large enough). Hence, the error on the tail $y > n^{\frac{1}{3}}$ is exponentially small in n. Thus, the expansion is valid throughout the whole range. □

Lemma 4.5 allows us to convert a line integral over a Beta function into one involving the Gamma function. The residues of the Gamma function are easier to evaluate. Therefore, we need to go back to a path integral. By Lemma 4.2, we are able to convert the integral in equation (4.22), where $\mathcal{C}$ encircles the poles at $\{a, \ldots, b\}$ and $f(n, z)$ has no singularities in the half plane $\Re(z) \geq a - \epsilon$, into a line integral along the line $\Re(z) = a - \epsilon$. By Lemma 4.5, we approximate the integral into a sum of line integrals over the Gamma function:

$$\frac{1}{2\pi\imath} \oint_{\mathcal{C}} f(n, z) \mathrm{B}(n + 1, -z) dz \sim \sum_{k,l} c_{k,l} \int_{a-\epsilon+\imath\infty}^{a-\epsilon-\imath\infty} f(n, z) n^{-k+z} \Gamma(-z + k) z^l dz \ . \tag{4.44}$$

To evaluate the integrals on the right hand side, again the residue theorem is used. The integrals on the right side are converted into closed-path integrals by showing that the missing parts of the path are "small".[1] Because we started out with the poles at $\{a, \ldots, b\}$ and assumed that there are no other singularities in the half plane $\Re(z) \geq a - \epsilon$, the line is usually completed into a box to the left around different poles. By Stirling's formula,[2] for $z = x + \imath y$ with $y \to \infty$ and bounded x, we have

$$|\Gamma(x + \imath y)| \sim \sqrt{\frac{2\pi}{|x + \imath y|}} e^{-x} |x + \imath y|^x e^{-y \arg(x+\imath y)} \sim \sqrt{2\pi} e^{-x} |y|^{x-\frac{1}{2}} e^{-\frac{\pi}{2}|y|} \ . \tag{4.45}$$

Thus, we can bound an integral for a path parallel to the x-axis for some $\alpha \leq \beta$ and $|m| > \max\{|\alpha|, |\beta|\}$ by

$$\begin{aligned} &\Big| \int_{\alpha \pm \imath m}^{\beta \pm \imath m} f(n, z) n^{-k+z} \Gamma(-z + k) z^l dz \Big| \\ &\qquad \leq c n^{r-k+\beta} (\beta + m)^l \sqrt{\frac{2\pi}{|x + \imath y|}} e^{-x} (\beta - \alpha) |\beta + m|^{\beta} e^{-m\frac{\pi}{4}} \xrightarrow[m \to \infty]{} 0 \ , \end{aligned} \tag{4.46}$$

for a suitable constant c.

To close the box, we need an integral parallel to the y-axis. We can bound the

[1] Derivations similar to the following can be found in [Knu98], Section 5.2.2.

[2] Which—in one of its many forms—is $\Gamma(z) = \sqrt{2\pi} z^{z-\frac{1}{2}} e^{-z} (1 + O(\frac{1}{|z|}))$.

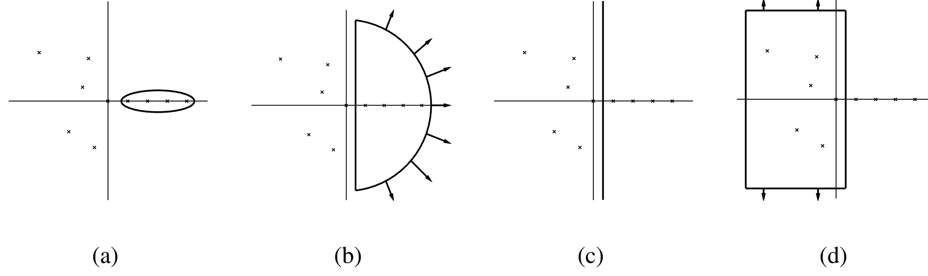


Figure 4.3: Illustration of the transformation of the integrals in the complex plane for the proof of Lemma 4.6. The Rice's integral (a) is transformed into a huge half-circle (b). The circle is extended to infinity, which leads to a line integral (c). The latter is extended into a box around different poles (d).

value of it for any constant α by

$$\left|\int_{\alpha-\imath\infty}^{\alpha+\imath\infty} f(n,z)n^{-k+z}\Gamma(-z+k)z^l dz\right| \leq cn^{r-k+\alpha}\int_{-\infty}^{+\infty}\left|\Gamma(k-\alpha+\imath y)(\alpha+y)^l\right|dy$$

$$\leq c'n^{r-k+\alpha}\int_{-\infty}^{+\infty}\sqrt{\frac{2\pi}{|k-\alpha+\imath y|}}e^{-k+\alpha}|k-\alpha+\imath y|^{k-\alpha}e^{-y\arg(k-\alpha+\imath y)}|\alpha+\imath y|^l dy$$

$$= O\left(n^{r-k+\alpha}\right) \quad (4.47)$$

because the integral converges to a constant due to the exponential decrease in $|y|$. As a result, choosing $\alpha < -r+k$ yields a sub-constant growth in n. Thus, we have just proven the following approximation lemma as a specialized and extended version of Rice's integrals (Theorem 2.2). The general idea is sketched in Figure 4.3.

Lemma 4.6 (Extended Rice's integrals).
Let $f(n,z)$ be an analytic continuation of $f(k,n) = f_{k,n}$ that contains the half line $[m,\infty)$. Assume further that for some $\epsilon > 0$ the function $f(n,z)$ has no singularities anywhere in the half plane $\Re(z) \geq m-\epsilon$ and that $f(z,n) = O(n^r)$ for some constant r (i.e., $f(n,z)$ is of constant growth with respect to z). Then for some constant N, we have

$$\sum_{k=m}^{n}(-1)^k\binom{n}{k}f_{k,n} = \frac{1}{2\pi}\sum_{k=0}^{N-1}\oint_{\mathcal{B}} f(n,z)\frac{(-1)^k B_k^{(1+z)}(1)}{k!}\Gamma(-z+k)n^{-k-z}dz$$

$$+ c\oint_{\mathcal{B}} f(n,z)n^{-N+z}\Gamma(-z)dz + o(1) \quad , \quad (4.48)$$

where $\mathcal{B}$ is a negatively oriented box of infinite height, right border at $m-\epsilon$, and left border $a < -r+N$ and c is an appropriate constant.

The general method to use Rice's integrals for sums exhibiting the exponential cancellation effect is not new. Almost the same approach is taken in [Szp88a] (Theorem 2.2). No proof is given for the validity of the approach to approximate the Beta

function with a Gamma function in [Szp88a]. Therefore, we derived the necessary bounds in this section. The two crucial points are the uniform expansion [Fie70] and the negligible tails.

A very early use of the method can be found in the third volume of the famous books by Knuth [Knu97a, Knu97b, Knu98]. The difference to our approach is that the Gamma function comes into play via a special identity for the exponential function.

4.3.3 The Average Compactification Number

In this section, we analyze the part labeled $\mathcal{A}_n$ of equation (4.21). It turns out that this has an interesting interpretation of its own. We call $\mathcal{A}_n$ the *average compactification number*. With "compactification" we want to describe the relation between the trie for the set S and the set itself. When the strings in S are inserted in the trie, common prefixes are merged. An edge starting at the root labeled a represents all strings in S that start with the character a. If there are n_a such strings, the edge "hides" $n_a - 1$ of these. Assume that we count all characters that we encounter by traversing the complete trie for S. Then these would be $\mathcal{A}_n$ less characters than $\|S\|$ (the number of characters in S). Hence, $n(d+1)/q - \mathcal{A}_n$ is an upper bound for the average performance of the TS algorithm. We give a brief derivation of the asymptotics for $\mathcal{A}_n$. This is somewhat redundant because the approach is very much the same and the derivation is much easier. We give the details nevertheless; they also serve as a primer to the technique.

We derive a recursive equation for $\mathcal{A}_n$ in analogy to Section 4.3.1. Again it is simpler to attach the counting to the nodes of the trie. At the node v with n leaves in the subtree, we have $n-1$ "hidden" characters in the edge leading to v. When counting this way, we overshoot the actual result by $n-1$. Hence, let $a_n = \mathcal{A}_n + n - 1$. We then have to sum over all possibilities to partition n (see equation (4.12)). For $n > 1$, this yields the formula

$$a_n = n - 1 + \sum_{i_1+\cdots+i_\sigma=n} \binom{n}{i_1, \ldots, i_\sigma} \sigma^{-n} \sum_{j=1}^{\sigma} a_{i_j} \ , \tag{4.49}$$

with $a_0 = 0$ and $a_1 = 0$ (because neither the empty trie nor the trie containing only one string hides any characters). The exponential generating function $A(z) = \sum_{n\geq 0} a_n \frac{z^n}{n!}$ is then recursively defined by

$$A(z) = ze^z - e^z + \sum_{i=1}^{\sigma} A\left(\frac{z}{\sigma}\right) e^{(1-\frac{1}{\sigma})z} + 1 \ . \tag{4.50}$$

Note that we again had to accommodate for the case $n = 0$ by adding one. Multiplying both sides by e^{-z} we get (with $\tilde{A}(z) = A(z)e^{-z}$)

$$\tilde{A}(z) = z - 1 + \sum_{i=1}^{\sigma} \tilde{A}\left(\frac{z}{\sigma}\right) + e^{-z} \ . \tag{4.51}$$

From the binomial transform (see equation (4.17)) we compute $\tilde{a}_0 = 0$ and $\tilde{a}_1 = 0$. For $n > 1$, we compare coefficients in equation (4.51) and find that

$$\tilde{a}_n = \frac{(-1)^n}{1 - \sigma^{1-n}} \ . \tag{4.52}$$

Translating this back, we get

$$a_n = \sum_{k=2}^{n} \binom{n}{k} \frac{(-1)^k}{1-\sigma^{1-k}} \quad . \tag{4.53}$$

This agrees with equation (4.21) because $\mathcal{A}_n = a_n - (n-1)$. The large terms in the sum can be as big as $|\binom{n}{n/2}\frac{(-1)^k}{1-\sigma^{1-k}}| \sim \frac{2^n}{\sqrt{n}}$. By the sign changes, these exponential values cancel out. This is often called an "exponential cancellation process". For these sums, Rice's integrals are the method of choice, and we use the results from the previous section. Because $\frac{1}{1-\sigma^{1-z}} = O(1)$ for $|z| \to \infty$, we can apply Lemma 4.6 to find that

$$a_n = \frac{1}{2\pi} \sum_{k=0}^{N-1} \oint_{\mathcal{B}} \frac{1}{1-\sigma^{1-z}} \frac{(-1)^k B_k^{(1+z)}(1)}{k!} \Gamma(-z+k) n^{-k+z} dz$$
$$+ c \oint_{\mathcal{B}} \frac{1}{1-\sigma^{1-z}} n^{-N+z} \Gamma(-z) dz + o(1) \quad , \tag{4.54}$$

where $\mathcal{B}$ is a box-path with the right border $\Re(z) = \frac{3}{2}$. Each term of the expansion is smaller by a factor of n and possibly more due to additional terms z in the generalized Bernoulli polynomials $B_k^{(1-z)}(1)$. To evaluate equation (4.54), we compute the residues left of the line $\Re(z) = \frac{3}{2}$. Because $\mathcal{B}$ is negatively oriented, we have to take the negative of the sum of the residues. We first consider the main term

$$\frac{1}{2\pi\imath} \oint_{\mathcal{B}} \frac{1}{1-\sigma^{1-z}} \Gamma(-z) n^z dz \quad . \tag{4.55}$$

The residues of $\frac{1}{1-\sigma^{1-z}}\Gamma(-z)n^z$ are

$$\text{res}\left[\frac{1}{1-\sigma^{1-z}}\Gamma(-z)n^z, z=0\right] = -\frac{1}{\sigma-1} \quad , \tag{4.56}$$

$$\text{res}\left[\frac{1}{1-\sigma^{1-z}}\Gamma(-z)n^z, z=1\right] = -\frac{n}{2} - \frac{\gamma-1}{\ln\sigma}n - n\log_\sigma n \quad , \tag{4.57}$$

and

$$\sum_{k\in\mathbb{Z}\setminus\{0\}} \text{res}\left[\frac{1}{1-\sigma^{1-z}}\Gamma(-z)n^z, z = 1 + \frac{2\pi\imath k}{\ln\sigma}\right]$$
$$= -\frac{n}{\ln\sigma} \sum_{k\in\mathbb{Z}\setminus\{0\}} n^{-\frac{2\pi\imath k}{\ln\sigma}} \Gamma\left(-1 + \frac{2\pi\imath k}{\ln\sigma}\right) \quad . \tag{4.58}$$

The sum in the last equation is a small oscillating term that is bounded by a small constant. These sums are frequently encountered throughout this type of problems.

The second term in the approximation (4.54) is $\frac{1}{1-\sigma^{1-z}}\Gamma(-z+1)n^{z-1}\frac{1-z}{2}$. Here, we have the residues

$$\text{res}\left[\frac{1}{1-\sigma^{1-z}}\Gamma(-z+1)n^{z-1}\frac{1-z}{2}, z=1\right] = \frac{1}{2\ln\sigma} \quad , \tag{4.59}$$

and

$$\sum_{k\in\mathbb{Z}\setminus\{0\}} \operatorname{res}\left[\frac{1}{1-\sigma^{1-z}}\Gamma(-z+1)n^{z-1}\frac{1-z}{2}, z=1+\frac{2\pi\imath k}{\ln\sigma}\right] = -\sum_{k\in\mathbb{Z}\setminus\{0\}} \frac{k\pi\imath}{(\ln\sigma)^2} n^{-\frac{2\pi\imath k}{\ln\sigma}}\Gamma\left(-1+\frac{2\pi\imath k}{\ln\sigma}\right) , \quad (4.60)$$

which are $O(1)$ together. Hence, we find (recall that we have to take the negative of the sum of all residues) that

$$a_n = n\log_\sigma n + n\left(\frac{1}{2}-\frac{1-\gamma}{\ln\sigma}+\frac{\sum_{k\in\mathbb{Z}\setminus\{0\}} n^{-\frac{2\pi\imath k}{\ln\sigma}}\Gamma\left(-1+\frac{2\pi\imath k}{\ln\sigma}\right)}{\ln\sigma}\right)+O(1) , \quad (4.61)$$

which proves the following lemma.

Lemma 4.7 (Asymptotic behavior of the compactification number).
The asymptotic behavior of $\mathcal{A}_n$ is

$$\mathcal{A}_n = n\log_\sigma n + n\left(-\frac{1}{2}-\frac{1-\gamma}{\ln\sigma}+\frac{\sum_{k\in\mathbb{Z}\setminus\{0\}} n^{-\frac{2\pi\imath k}{\ln\sigma}}\Gamma\left(-1+\frac{2\pi\imath k}{\ln\sigma}\right)}{\ln\sigma}\right)+O(1) . \quad (4.62)$$

4.3.4 Allowing a Constant Number of Errors

We now turn to the evaluation of the sum

$$\mathcal{S}_n^d = \sum_{k=2}^{n}\binom{n}{k}\frac{(-1)^k}{\sigma^{k-1}-1}\left(\frac{q}{\sigma^{k-1}-p}\right)^{d+1} , \quad (4.63)$$

at first for constant d. For this case, we apply Lemma 4.6. Hence,

$$\mathcal{S}_n^d = \frac{1}{2\pi}\sum_{k=0}^{N-1}\oint_{\mathcal{B}}\frac{1}{\sigma^{z-1}-1}\left(\frac{q}{\sigma^{z-1}-p}\right)^{d+1}\frac{(-1)^k B_k^{(1+z)}(1)}{k!}\Gamma(-z+k)n^{-k+z}dz + c\oint_{\mathcal{B}}\frac{1}{\sigma^{z-1}-1}\left(\frac{q}{\sigma^{z-1}-p}\right)^{d+1}n^{-N+z}\Gamma(-z)dz+o(1) , \quad (4.64)$$

where $\mathcal{B}$ is again a box-path with the right border $\Re(z)=\frac{3}{2}$. We concentrate on the main term of the approximation because the other terms are smaller by a factor of n or more (see also the previous section). By Lemma 4.6, $\mathcal{B}$ is negatively oriented. We simply change the orientation by substituting $-z$ for z. The integral considered for the remainder of this section is thus

$$\mathcal{I}_n^d = -\frac{1}{2\pi\imath}\int_{\mathcal{B}}\frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}dz , \quad (4.65)$$

where $\mathcal{B}$ is now a positively oriented box around all poles right of the line $\Re(z) = -\frac{3}{2}$. We evaluate equation (4.65) by its residues. Their locations in the complex plane are depicted in Figure 4.4. The poles are located at $z = -1$, $z = 0$, $z = -1 \pm \frac{2\pi\imath k}{\ln\sigma}$, and $z = -\log_\sigma p - 1 \pm \frac{2\pi\imath k}{\ln\sigma}$, $k \in \mathbb{Z}$. The real part of the latter ranges from $-1 + \log_\sigma(\sigma^2 - 1) > -1$ to one under the assumption that $\sigma^{-2} \le p \le 1 - \sigma^{-2}$. The latter is fulfilled if there can be at least one mismatch and at least one match (otherwise, the whole analysis becomes trivial anyway).

Lemma 4.8 (Residues at $z = -1 \pm \frac{2\pi\imath k}{\ln\sigma}$).
Let $g(z) := \frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}$. The residues of $g(z)$ at $z = -1 \pm \frac{2\pi\imath k}{\ln\sigma}$ are

$$\operatorname{res}\left[g(z), z = -1\right] = n\left(\frac{1-\gamma}{\ln\sigma} - \log_\sigma n + \frac{1}{2} + \frac{(d+1)}{q}\right) \tag{4.66}$$

and

$$\sum_{k\in\mathbb{Z}\setminus\{0\}} \operatorname{res}\left[g(z), z = -1 + \frac{2\pi\imath k}{\ln\sigma}\right] = \frac{1}{\ln\sigma}\sum_{k\in\mathbb{Z}\setminus\{0\}} -\Gamma\left(-1 + \frac{2\pi k\imath}{\ln\sigma}\right) n^{1-\frac{2\pi k\imath}{\ln\sigma}} . \tag{4.67}$$

Proof. The residue can be derived easiest from the series decompositions. These are at $z = -1 + \frac{2\pi\imath k}{\ln\sigma}$

$$\frac{1}{\sigma^{-1-z}-1} = \sum_{l=-1}^{\infty} \frac{(-\ln\sigma)^l B_{l+1}}{(l+1)!}\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right)^l \tag{4.68}$$

$$= \frac{-1}{\ln\sigma\left(z+1-\frac{2\pi\imath k}{\ln\sigma}\right)} - \frac{1}{2} \tag{4.69}$$

$$+ O\left(\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right)\right) \tag{4.70}$$

$$\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1} = 1 + \frac{(d+1)\ln\sigma}{q}\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right) \tag{4.71}$$

$$+ O\left(\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right)^2\right) \tag{4.72}$$

$$n^{-z} = n\sum_{l=0}^{\infty} n^{-\frac{2\pi\imath k}{\ln\sigma}} \frac{(-\ln n)^l}{l!}\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right)^l . \tag{4.73}$$

For the Gamma function, we have a pole at -1 but not at the other points $-1 + \frac{2\pi\imath k}{\ln\sigma}$. For $k \neq 0$, we have

$$\Gamma(z) = \sum_{l=-1}^{\infty} \gamma_l^{(-1)}(z+1)^l = \frac{-1}{z+1} + (\gamma - 1) + O\left((z+1)\right) \tag{4.74}$$

and, for $k = 0$, we have

$$\sum_{l=0}^{\infty} \gamma_l^{\left(-1+\frac{2\pi\imath k}{\ln\sigma}\right)}\left(z + 1 - \frac{2\pi\imath k}{\ln\sigma}\right)^l , \tag{4.75}$$

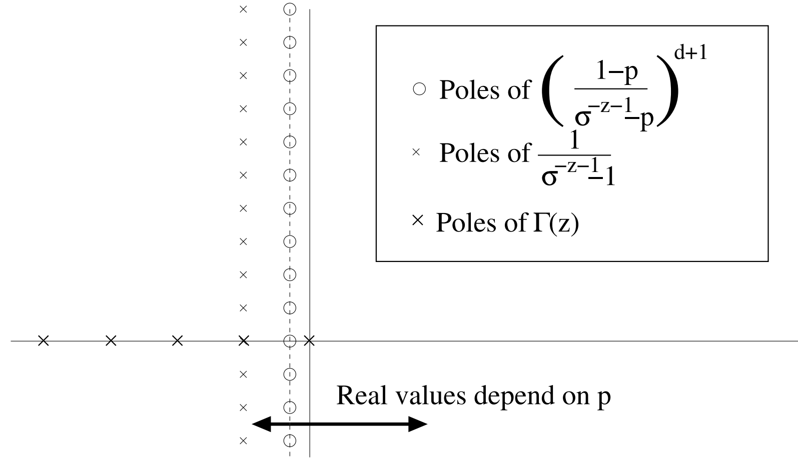


Figure 4.4: Location of poles of $\frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}$ in the complex plane.

where $\gamma_l^{x_0}$ is the l-th coefficient of the series expansion of the Gamma function at the point x_0. The B_l are the Bernoulli numbers (see equation (2.34)). Note that $\gamma_1^{(-1)} = \gamma \approx 0.577$ and $\gamma_0^{(-1+\frac{2\pi\imath k}{\ln\sigma})} = \Gamma\left(-1+\frac{2\pi k\imath}{\ln\sigma}\right)$. □

The next term in the approximation series in equation (4.64) includes the additional factor $n^{-1}B_1^{(1-z)}(1) = n^{-1}(-\frac{1-z}{2}+1)$. The factor z cancels any single pole, leaving only the simple pole at $z=-1$. The residues are derived in a similar manner from the series representations, but the order may be decreased by canceling a factor z, $z+1$, or similar. Furthermore, the results from the series decomposition are divided by the factor n, thus the residues are negligible.

Observe that

$$\operatorname{res}\left[g(z), z=-1\right] + \sum_{k\in\mathbb{Z}\setminus\{0\}} \operatorname{res}\left[g(z), z=-1+\frac{2\pi\imath k}{\ln\sigma}\right] = \underbrace{\frac{d+1}{q}n}_{\mathbb{E}\left[L_n^d\right]} - \underbrace{\left(n\log_\sigma n - \frac{1}{2}n - \frac{1-\gamma}{\ln\sigma}n - n\frac{\sum_{k\in\mathbb{Z}\setminus\{0\}} n^{-\frac{2\pi\imath k}{\ln\sigma}}\Gamma\left(-1+\frac{2\pi\imath k}{\ln\sigma}\right)}{\ln\sigma}\right)}_{-\mathcal{A}_n+O(1)}, \quad (4.76)$$

thus the residues at $\Re(z)=-1$ cancel the first terms in the total complexity of the TS algorithm (see equation (4.21) and Lemma 4.7). When allowing a constant number of errors, the complexity depends upon the residues at $-\log_\sigma p - 1 \pm \frac{2\pi\imath k}{\ln\sigma}$ and at zero. If $-\log_\sigma p - 1 \neq 0$ then there is a single residue due to the Gamma function at zero.

Lemma 4.9 (Residue at $z=0$).
Let $g(z) := \frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}$. *If* $-\log_\sigma p - 1 \neq 0$ *the residue of* $g(z)$ *at* $z=0$ *is*

$$-\operatorname{res}\left[g(z), z=0\right] = \frac{\sigma}{\sigma-1}\left(\frac{q\sigma}{1-p\sigma}\right)^{d+1}. \quad (4.77)$$

Proof. The residue at this single pole is again evaluated by the series decomposition. For the Gamma function, the coefficient of the z^{-1}-term is one. The other terms are just the function values at $z = 0$. The residue is the product of these and results in equation (4.77). □

The remaining residues at $-\log_\sigma p - 1 \pm \frac{2\pi\imath k}{\ln\sigma}$ (including the case $z = 0$ for $-\log_\sigma p - 1 = 0$) are computed in the next step.

Lemma 4.10 (Residues at $z = 0, z = -\log_\sigma p - 1 + \frac{2\pi\imath k}{\ln\sigma}$).
Let $g(z) := \frac{1}{\sigma^{-1-z}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}$. *If we have* $p = \sigma^{-1}$, $g(z)$ *has the residue*

$$\operatorname{res}\left[g(z), z = 0\right] = \sum_{l=0}^{d+1} \frac{(\sigma-1)^{d+1}(-1)^{-l}B_{d+1}^{(-l+d+1)}}{(-l+d+1)!} \cdot \sum_{i=0}^{l}\left[\sum_{j=0}^{i}\frac{A_j\left(\sigma^{-1}\right)}{j!(i-j)!}\left(\frac{\sigma}{1-\sigma}\right)^{j+1}\left(-\frac{\ln n}{\ln\sigma}\right)^{i-j}\right]\frac{\gamma_{l-i-1}^{(0)}}{(\ln\sigma)^{l-i}} \quad (4.78)$$

at $z = 0$. *For* $k \in \mathbb{Z}$ *and* $k \neq 0$ *or* $p \neq \sigma^{-1}$, $g(z)$ *has the residues*

$$\operatorname{res}\left[g(z), z = -\log_\sigma p - 1 + \frac{2\pi\imath k}{\ln\sigma}\right] = \sum_{l=0}^{d}\left(\frac{1-p}{p}\right)^{d+1}\frac{(-\ln\sigma)^{-l}B_{-l+d}^{(d+1)}}{(-l+d)!} \cdot \sum_{i=0}^{l}\left[\sum_{j=0}^{i}\frac{-A_j(p)(-\ln\sigma)^j}{(1-p)^{j+1}j!}n^{\log_\sigma p+1-\frac{2\pi\imath k}{\ln\sigma}}\frac{(-\ln n)^{i-j}}{(i-j)!}\right]\gamma_{l-i}^{\left(-\log_\sigma p-1+\frac{2\pi\imath k}{\ln\sigma}\right)} . \quad (4.79)$$

The $B_k^{(a)}$ *are generalized Bernoulli numbers, the* $A_l(x)$ *are Eulerian polynomials, and the* $\gamma_l^{(x_0)}$ *are the coefficients of the series for* $\Gamma(z)$ *at* $z = x_0$.

Proof. We compute the residues at $z = 0$ and $z = -\log_\sigma p - 1 + \frac{2\pi\imath k}{\ln\sigma}$. If $p \neq \sigma^{-1}$, then the residue at $z = 0$ is given by equation (4.77). Otherwise we have a higher order pole at $z = 0$. Therefore, we need to use a different series expansion for the term $(\frac{q}{\sigma^{-z-1}-p})^{d+1}$. We find the following series representations. The pole is of order $d+1$ or $d+2$ (depending on whether $p = \sigma^{-1}$) so we only give the abstract sums and do not bother with evaluating the first terms (as in the proof of Lemma 4.8). Let $a_k = \log_\sigma p + 1 - \frac{2\pi\imath k}{\ln\sigma}$.

$$\frac{1}{\sigma^{-1-z}-1} = \sum_{l=0}^{\infty}\frac{-A_l(p)(-\ln\sigma)^l}{(1-p)^{l+1}l!}(z+a_k)^l \quad (4.80)$$

$$\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1} = \sum_{l=-d-1}^{\infty}\left(\frac{1-p}{p}\right)^{d+1}\frac{(-\ln\sigma)^l B_{l+d+1}^{(d+1)}}{(l+d+1)!}(z+a_k)^l \quad (4.81)$$

$$n^{-z} = \sum_{l=0}^{\infty} n^{1+\log_\sigma p}n^{-\frac{2\pi\imath k}{\ln\sigma}}\frac{(-\ln n)^l}{l!}(z+a_k)^l . \quad (4.82)$$

For the Gamma function, we have a pole at zero but not at the other points $z = -a_k$. For $p = \sigma^{-1}$ and $k = 0$, we have

$$\Gamma(z) \quad = \quad \sum_{l=-1}^{\infty} \gamma_l^{(0)} z^l \quad , \tag{4.83}$$

and, for $k \neq 0$ or $p \neq \sigma^{-1}$, we have

$$\Gamma(z) \quad = \quad \sum_{l=0}^{\infty} \gamma_l^{(-a_k)} (z + a_k)^l \quad . \tag{4.84}$$

The $\gamma_l^{(-a_k)}$ are just the result of a simple Taylor series. The relevant poles are of order $d + 1$ (or $d + 2$ for $p = \sigma^{-1}$). The residue is the sum of all combinations of coefficients such that the power of $(z + a_k)$ is -1. The series lead directly to the residues. □

Remark 4.11.
One can show that $|\gamma_l^{(0)} - (-1)^l| < \left(\frac{1}{2} + \epsilon\right)^l$ as follows: By subtracting the poles at zero and -1 from $\Gamma(z)$, we get a function $\Gamma(z) - \frac{1}{z} + \frac{1}{z+1}$ which is analytical for $|z| < 2$. Because $-\frac{1}{z} + \frac{1}{z+1} = -\frac{1}{z} - \sum_{l=0}^{\infty} (-1)^l z^l$, we know that the series $\sum_{l=0}^{\infty} (\gamma_l^{(0)} - (-1)^l) z^l$ is convergent with a radius of convergence of two, which proves our claim.

We can now determine the exact complexity for any constant value of d by evaluating the sum (possible with the help of a computer algebra system). The number of terms in equations (4.78) and (4.79) is polynomial in d, so the sums are more "efficient" than equation (4.20). For most purposes, the term with the largest growth in n should suffice. If $p = \sigma^{-1}$, this term is

$$-\frac{\sigma(\sigma - 1)^d}{(d + 1)!} (\log_\sigma n)^{d+1} \quad . \tag{4.85}$$

Otherwise this term is

$$-\frac{(1 - p)^d}{d! p^{d+1}} (\log_\sigma n)^d n^{1+\log_\sigma p} \sum_{k \in \mathbb{Z}} n^{-\frac{2\pi \imath k}{\ln \sigma}} \Gamma\left(-\log_\sigma p - 1 + \frac{2\pi \imath k}{\ln \sigma}\right) \quad . \tag{4.86}$$

The infinite sum is somewhat dissatisfactory, but it is bounded by a constant (due to the behavior of the Gamma function along an imaginary line), and these sums do occur in a lot of problems of this kind. Note that the negative sign cancels with the sign in equation (4.65). The result is summarized in the following theorem.

Theorem 4.12 (Searching with a constant bound).
Let $d = O(1)$, then

$$\mathbb{E}\left[T_n^d\right] = \begin{cases} O\left((\log n)^{d+1}\right) & \textit{for } p = \sigma^{-1} \ , \\ O\left((\log_\sigma n)^d n^{1+\log_\sigma p}\right) & \textit{for } p > \sigma^{-1} \ , \\ O(1) & \textit{otherwise} \ . \end{cases} \tag{4.87}$$

Note that $1 + \log_\sigma p < 1$ for $p < 1$. Thus, for a constant number of mismatches and almost any model, the TS algorithm is more efficient than the LS algorithm.

4.3.5 Allowing a Logarithmic Number of Errors

Whereas for small d it is still feasible to evaluate the sum in Lemma 4.10, it is not satisfactory for $d = f(n)$ as a function of n. In this section, we also answer the question for which choice of d the LS algorithm is asymptotically as efficient as the TS algorithm (on average). The sums in Lemma 4.10 are rather hard to evaluate for growing d. We look at the integral (4.65) as a line integral. Instead of computing the residues, we bound the integral, possibly first sweeping over some residues. This technique has a strong resemblance to the application of the Mellin transform although we used Rice's integrals (see also the remarks in [Szp88a] and [FS95]).

The integral $\mathcal{I}_n^d$ defined in equation (4.65) gives the highest order term of the sum $\mathcal{S}_n^d$ of equation (4.63). By equations (4.46) and (4.47) we find that

$$\mathcal{I}_n^d = \mathcal{I}_{\xi,n}^d = \frac{1}{2\pi\imath}\int_{\xi-\imath\infty}^{\xi+\imath\infty} \frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{d+1}\Gamma(z)n^{-z}dz \ , \tag{4.88}$$

where $\xi = -\frac{3}{2}$. In fact, we can move the line of integration freely between -2 and -1 because the function under the integral is analytic in this strip. When taking the residues at $\Re(z) = -1$ and possibly at $\Re(z) = 0$ into account we can move the line of integration even further. Thus, we have (taking into account equation (4.76))

$$\mathcal{I}_n^d = \begin{cases} \mathcal{I}_{\xi,n}^d & \text{, for } -2 < \xi < -1 \\ \mathcal{I}_{\xi,n}^d + \mathbb{E}\left[L_n^d\right] - \mathcal{A}_n + O(1) & \text{, for } -1 < \xi < -1 - \log_\sigma p \le 0 \\ \mathcal{I}_{\xi,n}^d + \mathbb{E}\left[L_n^d\right] - \mathcal{A}_n & \\ \quad + \frac{\sigma}{\sigma-1}\left(\frac{q\sigma}{1-p\sigma}\right)^{d+1} + O(1) & \text{, for } 0 < \xi < -1 - \log_\sigma p \ . \end{cases} \tag{4.89}$$

In the following, we show that $\mathcal{I}_{\xi,n}^d$ is sublinear for certain choices of ξ depending on q, p, and d. To this end, we assume that $d+1 = c\log_\sigma n$ and bound the integral $\mathcal{I}_{\xi,n}^d$ as an exponent of n. We find that

$$\begin{aligned} \left|\mathcal{I}_{\xi,n}^d\right| &= \left|\frac{1}{2\pi\imath}\int_{\xi-\imath\infty}^{\xi+\imath\infty} \frac{1}{\sigma^{-z-1}-1}\left(\frac{q}{\sigma^{-z-1}-p}\right)^{c\log_\sigma n}\Gamma(z)n^{-z}dz\right| \\ &\le \frac{1}{2\pi}\frac{1}{\sigma^{-\xi-1}-1}\left(\frac{q}{\sigma^{-\xi-1}-p}\right)^{c\log_\sigma n} n^{-\xi}\int_{\xi-\imath\infty}^{\xi+\imath\infty}|\Gamma(z)|dz \\ &= O\left(n^{-\xi+c\log_\sigma\left(\frac{q}{\sigma^{-\xi-1}-p}\right)}\right) \ , \end{aligned} \tag{4.90}$$

because p and one are real values, the maxima of $|\frac{1}{\sigma^{-z-1}-1}|$ and $|\frac{q}{\sigma^{-z-1}-p}|$ are taken for $\arg z = 0$, i.e., $z = \xi$. This can easily be seen as follows. The maximum is taken for the minimum of the denominator. The denominator describes a circle with center -1 or $-p$ and radius $\sigma^{-\xi-1}$ in the complex plane. The closest distance to zero is then attained at the intersection with the real line left of the center. Let

$$\mathcal{E}_{c,p,\xi} = c\log_\sigma\left(\frac{1-p}{\sigma^{-\xi-1}-p}\right) - \xi \tag{4.91}$$

be the exponent. We can bound it as follows.

Lemma 4.13.
For $0 < p < 1$, $c \geq 0$, and $c \neq 1 - p$, there exists a $\xi > -2$ such that $\mathcal{E}_{c,p,\xi} < 1$. If $c < 1$, then $\mathcal{E}_{c,p,\xi}$ has a minimum at $\xi^* = \log_\sigma (1 - c) - \log_\sigma p - 1$. If $c \geq 1$ or $\xi^* \leq -2$, then some value $\xi \in (-2, -1)$ satisfies $\mathcal{E}_{c,p,\xi} < 1$.

Proof. We examine the growth of $\mathcal{E}_{c,p,\xi}$ with respect to ξ. The derivatives are

$$\frac{d}{d\xi}\mathcal{E}_{c,p,\xi} = -1 + \frac{c}{1 - p\sigma^{\xi+1}} \quad \text{and} \quad \frac{d^2}{d\xi^2}\mathcal{E}_{c,p,\xi} = \frac{cp\sigma^{\xi+1} \ln \sigma}{(1 - p\sigma^{\xi+1})^2} \ , \tag{4.92}$$

which shows that, for $c > 0$, the exponent has at most one local extreme value for real ξ at $\xi^* = \log_\sigma (1 - c) - \log_\sigma p - 1$, which is a minimum with value

$$c \log_\sigma \left(\frac{q(1 - c)}{pc} \right) + \log_\sigma \left(\frac{p}{1 - c} \right) + 1 \ . \tag{4.93}$$

Assume $c < 1$ and let $c = 1 - \sigma^{-x}$, then the exponent has a minimum at $\xi^* = -x - 1 - \log_\sigma p$, where it takes the value

$$\left(1 - \sigma^{-x}\right) \left(\log_\sigma \left(\frac{q}{p} \right) - \log_\sigma (\sigma^x - 1) \right) + x + 1 + \log_\sigma p \ . \tag{4.94}$$

With respect to x, the exponent has derivative

$$\sigma^{-x} \ln \sigma \left(\log_\sigma \left(\frac{q}{p} \right) - \log_\sigma (\sigma^x - 1) \right) \quad \begin{cases} > 0 & \text{, for } x < -\log_\sigma p \\ = 0 & \text{, for } x = -\log_\sigma p \\ < 0 & \text{, for } x > -\log_\sigma p \ . \end{cases} \tag{4.95}$$

Hence, there is a single extreme value, a maximum at $x^* = -\log_\sigma p$, where the exponent takes the value one. For all other values of x, the exponent is smaller than one. Thus, for $\xi^* > -2$, the exponent is smaller than one if $x \neq -\log_\sigma p$, which is equivalent to $c \neq 1 - p$. This proves our claim for $\xi^* > -2$ and $c < 1$.

For $\xi^* \leq -2$, we find that $c > 1 - \frac{p}{\sigma}$. The minimum value of the exponent is taken for some $\epsilon > 0$ at $\xi = -2 + \epsilon$. Because, for small ϵ, we have

$$c \log_\sigma \left(\frac{1 - p}{\sigma^{1-\epsilon} - p} \right) < c \log_\sigma 1 = 0 \ . \tag{4.96}$$

Thus, we find that

$$\mathcal{E}_{c,p,-2+\epsilon} < \left(1 - \frac{p}{\sigma}\right) \log_\sigma \left(\frac{1 - p}{\sigma^{1-\epsilon} - p} \right) + 2 - \epsilon \ . \tag{4.97}$$

The derivative of the right hand side of equation (4.97) with respect to p is

$$\frac{-1}{\sigma} \log_\sigma \left(\frac{1 - p}{\sigma^{1-\epsilon} - p} \right) + \left(1 - \frac{p}{\sigma}\right) \left(-\frac{\sigma^{1-\epsilon} - 1}{(\sigma^{1-\epsilon} - p)(1 - p) \ln \sigma} \right). \tag{4.98}$$

The second derivative is

$$\frac{2}{\sigma} \left(\frac{\sigma^{1-\epsilon} - 1}{(\sigma^{1-\epsilon} - p)(1 - p) \ln \sigma} \right) - \left(1 - \frac{p}{\sigma}\right) \left(\frac{(\sigma^{1-\epsilon} - 1) \left((1 - p) + (\sigma^{1-\epsilon} - p)\right)}{(\sigma^{1-\epsilon} - p)^2 (1 - p)^2 \ln \sigma} \right) \ , \tag{4.99}$$

which is smaller than zero for

$$p < \frac{\sigma^{2-\epsilon} + \sigma - 2\sigma^{1-\epsilon}}{-1 + 2\sigma - \sigma^{1-\epsilon}} \xrightarrow[\epsilon \to 0]{} \sigma \ . \tag{4.100}$$

Therefore, the first derivative is decreasing for $p < \sigma - \epsilon$, and it is maximal at $p = 0$. Here we have a value of

$$\frac{(1-\epsilon)\ln\sigma - \sigma + \sigma^{\epsilon}}{\sigma \ln \sigma} \xrightarrow[\epsilon \to 0]{} \frac{\ln\sigma - \sigma + 1}{\sigma \ln\sigma} \ , \tag{4.101}$$

which is smaller than zero for $\sigma \geq 2$. Therefore, the right hand side of equation (4.97) is decreasing in p, and the maximal value is attained at $p = 0$, where we have a value of one. Because $p < 1$, we have an exponent of $\mathcal{E}_{c,p,-2+\epsilon} < 1$ for some small $\epsilon > 0$.

Finally, for $c \geq 1$ there is no minimum value ξ^*. The derivative of the exponent $\mathcal{E}_{c,p,\xi}$ with respect to ξ is

$$\frac{c\sigma^{-\xi-1}}{\sigma^{-\xi-1} - p} - 1 \geq \frac{1}{1 - p\sigma^{\xi+1}} - 1 \ , \tag{4.102}$$

which is strictly positive for $\xi > -1 - \log_\sigma p$. Thus, the minimum is attained at the smallest possible value for ξ in the interval $(-2, \infty)$, say at $\xi = -2 + \epsilon$. Again the term $\log_\sigma \left(\frac{1-p}{\sigma^{1-\epsilon}-p}\right)$ is negative so that

$$\mathcal{E}_{c,p,1-\epsilon} = c\log_\sigma\left(\frac{1-p}{\sigma^{1-\epsilon} - p}\right) + 2 - \epsilon \leq \log_\sigma\left(\frac{1-p}{\sigma^{1-\epsilon} - p}\right) + 2 - \epsilon \ . \tag{4.103}$$

With respect to p, the right hand side has the derivative

$$-\frac{\sigma^{1-\epsilon} - 1}{(\sigma^{1-\epsilon} - p)(1-p)\ln\sigma} < 0 \ , \tag{4.104}$$

so the maximal value is again obtained at $p = 0$, where we already know that the exponent has a value of one. Hence, we have $\mathcal{E}_{c,p,-2+\epsilon} < 1$ for $p > 0$. This finishes the proof. □

The bounds immediately yield the following theorem.

Theorem 4.14 (Searching with a logarithmic bound).
If $d + 1 = c\log_\sigma n$, then we have

$$\mathbb{E}\left[T_n^d\right] = \begin{cases} o(n), & \text{for } c < q \\ \Theta(n \log n), & \text{for } c > q \ . \end{cases} \tag{4.105}$$

Proof. By Lemma 4.13 and equation (4.90), we can bound $\mathcal{I}^d_{\xi,n}$. The minimum of the exponent—if it exists—is at $\xi^* = \log_\sigma(1-c) - \log_\sigma p - 1 < -\log_\sigma p - 1$. We make a case distinction on ξ^*.

Case 1. The minimum ξ^* exists and $\xi^* > -1$. Here, we move the line of integration to $\xi = \xi^*$. By Lemma 4.13, we have $\mathcal{I}^d_{\xi,n} = o(n)$. By equation (4.89), we find that

$$\mathbb{E}\left[T_n^d\right] = o(n) \ , \tag{4.106}$$

if we can bound the term

$$\frac{\sigma}{\sigma-1}\left(\frac{q\sigma}{1-p\sigma}\right)^{d+1} \tag{4.107}$$

for $\xi^* > 0$ by $o(n)$. Note that $\xi^* > 0$ implies $c < 1 - p\sigma$ and also $p < \sigma^{-1}$ because $c > 0$. We bound equation (4.107) by

$$\frac{\sigma}{\sigma-1}\left(\frac{q\sigma}{1-p\sigma}\right)^{d+1} = \frac{\sigma}{\sigma-1} n^{c\log_\sigma\left(\frac{1-p}{\sigma^{-1}-p}\right)} < \frac{\sigma}{\sigma-1} n^{(1-p\sigma)\log_\sigma\left(\frac{1-p}{\sigma^{-1}-p}\right)} . \tag{4.108}$$

The exponent $(1-p\sigma)\log_\sigma\left(\frac{1-p}{\sigma^{-1}-p}\right)$ can be bounded by one from above: Its derivative with respect to p is

$$-\sigma\log_\sigma\left(\frac{1-p}{\sigma^{-1}-p}\right) + \sigma\frac{1-\sigma^{-1}}{(1-p)\ln\sigma} , \tag{4.109}$$

which is strictly less than zero because $\frac{1-p}{\sigma^{-1}-p} > \sigma$ and $\frac{1-\sigma^{-1}}{1-p} < 1$ (note again that $p < \sigma^{-1}$). At $p = 0$, the exponent has value one. Hence, it must be smaller for $p > 0$.

Case 2. The minimum ξ^* exists and $-2 < \xi^* < -1$. We move the line of integration to ξ^* and find that

$$\mathbb{E}\left[T_n^d\right] = \mathbb{E}\left[L_n^d\right] - \mathcal{A}_n + o(n) = \Theta(n\log n) \tag{4.110}$$

because $\xi^* < -1$ implies $c > q$, and so

$$\begin{aligned}\mathbb{E}\left[L_n^d\right] - \mathcal{A}_n &= n\left(\frac{c\log_\sigma n}{q} - \log_\sigma n + O(1)\right) \\ &= \epsilon n\log_\sigma n + O(n) = \Theta(n\log n)\end{aligned} \tag{4.111}$$

for $\epsilon = \frac{c}{q} - 1$.

Case 3. No minimum ξ^* exists or the minimum exists with $\xi^* < -2$. In this case, we have either $c > 1$ or $\xi^* < -2$, and we move the line of integration to a value $-2 < \xi < -1$ where, by Lemma 4.13, we find that $\mathcal{I}_{\xi,n}^d = o(n)$. Because $\xi^* < -2$ implies $c > 1 - p\sigma^{-1} > q$ and the non-existence of a minimum implies $c > 1 > q$, equation (4.110) and as a consequence equation (4.111) also hold.

As a result, we have a linear-logarithmic behavior in Cases 2 and 3 and a sublinear behavior in Case 1. The latter occurs if and only if $\xi^* > -1$, which is equivalent to $c > q$.

Figure 4.5 illustrates the proof idea. □

4.3.6 Remarks on the Complexity of the TS Algorithm

To guide the selection between the two algorithms and to improve the understanding of the TS algorithm, we detail the exponent ϵ in the sublinear growth term $O(n^\epsilon) = o(n)$ and the influence of the error model.

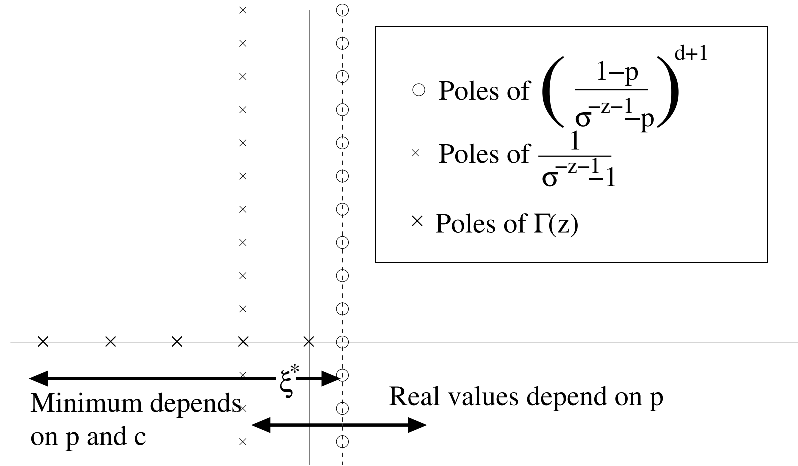


Figure 4.5: Illustration for Theorem 4.14.

4.3.6.1 Explicit Computation of the Exponent

It is possible to determine concrete values for the exponent of n when $\mathbb{E}[T_n^d]$ has sub-linear growth. This is the case for $0 < c < 1-p$. Either by Lemma 4.13 or by Lemma 4.9, the complexity is bounded by $O(n^{\mathcal{E}_{c,p,\min(\xi^*,0)}})$ where

$$\mathcal{E}_{c,p,\min(\xi^*,0)} = \begin{cases} \log_\sigma \left((1-c)^{c-1}c^{-c}\sigma p^{1-c}(1-p)^c\right) & \text{for } 1-p\sigma < c < 1-p \ , \\ \log_\sigma \left((1-p)^c\sigma^c(1-p\sigma)^{-c}\right) & \text{for } 0 \le c \le 1-p\sigma \ . \end{cases} \tag{4.112}$$

The first line is from Lemma 4.13, the second, for the case $\xi^* > 0$, follows from Lemma 4.9. The second term is larger than the first because

$$\begin{aligned} \log_\sigma \left((1-p)^c\sigma^c(1-p\sigma)^{-c}\right) - \log_\sigma \left((1-c)^{c-1}c^{-c}\sigma p^{1-c}(1-p)^c\right) \\ = c\log_\sigma\left(\frac{p\sigma c}{(1-p\sigma)(1-c)}\right) + \log_\sigma\left(\frac{1-c}{\sigma p}\right) \end{aligned} \tag{4.113}$$

has value zero at $c = 1-p\sigma$ and derivative

$$\log_\sigma\left(\frac{p\sigma c}{(1-p\sigma)(1-c)}\right) \begin{cases} > 0 & \text{for } c > 1-p\sigma \ , \\ = 0 & \text{for } c = 1-p\sigma \ , \\ < 0 & \text{for } c < 1-p\sigma \ . \end{cases} \tag{4.114}$$

Thus, there is a local minimum at $c = 1-p\sigma$. On the other hand, the second term is hidden in the first for $c > 1-p\sigma$.

The exponent is growing in p (which should not be surprising). The first term has derivative $\frac{1-c}{p} - \frac{c}{1-p}$ with respect to p, which is positive for $c < 1-p$. The second term's derivative with respect to p is $\frac{-c}{1-p} + \frac{\sigma c}{1-\sigma p}$, which is always positive.

For $p = \sigma^{-1}$ (Hamming distance), the exponent is $\frac{H(c)}{\ln\sigma} + c\log_\sigma(\sigma-1)$, where $H(x) = -x\ln(x) - (1-x)\ln(1-x)$ is the entropy function. The exponent is zero at

$c = 0$ and one at $c = 1 - p$. Thus, for $\sigma = 2$, we have a scaled entropy function. With growing σ, the linear part dominates more and more while the entropy part diminishes, i.e., the behavior of the exponent converges to a linear function. As a result, we can estimate that the running time can be bounded by a function between $O(n^{H(c)/\ln 2})$ and $O(n^c)$.

4.3.6.2 Relation between the Error Probability and the Number of Errors

For the LS algorithm, we can double the number of allowed errors and the error probability and get the same asymptotic running time. For the TS algorithm, in the parameter range with sublinear behavior, this is different. Here, a higher variance in the length that a string is matched by the pattern increases the expected running time. This behavior is explained by the difference in savings or expenses of an early or late end in matching a pattern against a string. Because of the trie structure, characters at the beginning of the strings are more compacted (see Section 4.3.3) whereas characters at the end are less compacted. Thus, not comparing some characters early in the string in exchange for comparing some characters more towards the end of another string is disadvantageous. The savings in the earlier characters correspond to fewer nodes than the extra cost in the characters towards the end. In our formula for the exponent given by equation (4.112), this can be seen when replacing c by ct and q by qt:

$$\mathcal{E}_{ct,1-tq,\min(\xi^*,0)} = \begin{cases} \log_\sigma\left(\frac{\sigma(1-tq)^{1-tc}(tq)^{tc}}{(1-tc)^{1-tc}(tc)^{tc}}\right), & \text{for } 1-(1-tq)\sigma \leq c < tq \\ tc\log_\sigma\left(\frac{tq\sigma}{1-(1-tq)\sigma}\right), & \text{for } 0 \leq c < 1-(1-tq)\sigma \end{cases} . \tag{4.115}$$

The derivative (with respect to t) of the above is always negative when t, c, and q result in valid parameters. For the second line, we have the derivative

$$\underbrace{c\log_\sigma\left(\frac{tq\sigma}{1-\sigma+tq\sigma}\right)}_{\geq 0} + \underbrace{\frac{tc}{\ln\sigma}}_{\geq 0}\underbrace{\frac{1-\sigma+tq\sigma}{tq\sigma}}_{\geq 1}\underbrace{\frac{\overbrace{q\sigma}^{\geq 0}\overbrace{(1-\sigma)}^{<0}}{(1-\sigma+tq\sigma)^2}}_{\geq 0} < 0 \ . \tag{4.116}$$

For the first line, we find that the derivative is

$$c\log_\sigma\left(\frac{q-tqc}{c-tqc}\right) + \frac{c-q}{(1-tq)\ln\sigma} \ , \tag{4.117}$$

which is negative if

$$c\log_\sigma(q-tqc) - \frac{q}{(1-tq)\ln\sigma} < c\log_\sigma(c-tqc) - \frac{c}{(1-tq)\ln\sigma} \ . \tag{4.118}$$

The function

$$x \to c\log_\sigma(x-tqc) - \frac{x}{(1-tq)\ln\sigma} \tag{4.119}$$

decreases for $c \leq x \leq q$ because the derivative with respect to x is

$$\frac{c}{(x-tqc)\ln\sigma} - \frac{c}{(c-tqc)\ln\sigma} \ , \tag{4.120}$$

which is negative or zero because $c \leq x$ and $1 - tq > 0$.

4.3.6.3 Interpretation

In summary, the average-case complexity of the TS algorithm is

$$\mathbb{E}\left[T_n^d\right] = \begin{cases} O\left((\log n)^{d+1}\right), & \text{for } d = O(1) \text{ and } p = \sigma^{-1} \\ O\left((\log n)^d n^{1+\log_\sigma p}\right), & \text{for } d = O(1) \text{ and } p > \sigma^{-1} \\ O(1), & \text{for } d = O(1) \text{ and } p < \sigma^{-1} \\ o(n), & \text{for } d+1 < q\log_\sigma n \\ \Theta(dn) = \Omega(n\log_\sigma n), & \text{for } d+1 > q\log_\sigma n \end{cases} \tag{4.121}$$

It is well known that the average height of a trie is asymptotically equal to $\log_\sigma n$ (see, e.g., [Pit86, Szp88b]). When no more branching takes place, the TS algorithm and the LS algorithm behave the same; both algorithms perform a constant number of comparisons on average. If we allow enough errors to go beyond the height of the trie, they should perform similar. With an error probability of q we expect to make qm errors on m characters. It is therefore not a big surprise that there is a threshold at $d+1 = q\log_\sigma n$.

With respect to the matching probability p, we have a different behavior for the three cases $p < \sigma^{-1}$, $p = \sigma^{-1}$, and $p > \sigma^{-1}$. To explain this phenomenon, we look at the conditional probability of a match for an already chosen character. If we have $p < \sigma^{-1}$, then the conditional probability must be smaller than one, i.e., with some probability independent of the pattern we have a mismatch and thus restrict the search independently of the pattern. If we have $p > \sigma^{-1}$, the conditional probability must be greater than one. Hence, with some probability independent of the pattern character we have a match and thereby extend our search. This restriction or extension is independent of the number of errors allowed; hence, there is the additional factor of n^ϵ in the complexity.

It is interesting to look at some concrete and important examples of comparison-based string distances. We already mentioned the Hamming distance and its extension with don't-care symbols in Section 2.1.3. Another model used, e.g., in [BT03, BTG03] is a kind of arithmetic distance. Here, we assume that $\Sigma = [0, \ldots, \sigma - 1]$ is ordered and that $k < \frac{\sigma-1}{2}$ is a constant. For $i, j \in \Sigma$ let $a(i,j) = \max(i,j) - \min(i,j)$. Then let the *arithmetic distance* be defined by

$$\mathrm{d}(i,j) = \begin{cases} 1 & \text{if } \min\left(a(i,j), \sigma - a(i,j)\right) > k \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{4.122}$$

For example, let Σ be the discretization of all angles between zero and 2π, then $\mathrm{d}(i,j)$ measures whether two angles are not "too far apart".

The match and mismatch parameters are given in Table 4.6. Especially for constant size d, we find that the introduction of don't cares makes a huge difference in the complexity, i.e., the complexity jumps from $O(\log^{d+1} n)$ to $O(n^{\log_\sigma(3-2/\sigma)}\log^d n)$. For logarithmically growing d, we already computed the exponent for the Hamming distance to be $H(c)/\ln\sigma + c\log_\sigma(\sigma-1)$, where $H(x) = -x\ln(x) - (1-x)\ln(1-x)$ is the entropy function. The introduction of don't-care symbols changes the exponent to $H(c)/\ln\sigma + c\log_\sigma(\sigma^2 - 3\sigma + 2) - c\log_\sigma(3\sigma - 2) + \log_\sigma(3 - \frac{2}{\sigma})$, i.e., a slightly smaller linear term in c but with the additional term $\log_\sigma(3 - \frac{2}{\sigma})$. Note also that the range for that we have sublinear growth is larger for the Hamming distance.

Distance	p	q
Hamming distance	$\frac{1}{\sigma}$	$\frac{\sigma-1}{\sigma}$
Hamming distance with don't care symbols	$\frac{3\sigma-2}{\sigma^2}$	$\frac{\sigma^2-3\sigma-2}{\sigma^2}$
Arithmetic distance	$\frac{2i+1}{\sigma}$	$\frac{\sigma-2i-1}{\sigma}$

Figure 4.6: Match and mismatch probabilities for selected comparison-based string distances.

4.4 Applications

Our research was also motivated by a project for the control of animal breeding via single nucleotide polymorphisms (SNPs) [WDH$^+$04]. An SNP is a location on the DNA sequence of a species that varies among the population but is stable through inheritance. In [WDH$^+$04], a sequence of SNPs is encoded as a string to identify individuals. In particular, the SNPs have one of the types “heterozygous”, “homozygous 1”, “homozygous 2”, and “assay failure”, which is encoded in an alphabet of size four. Because of errors, the search in the dataset of all individuals needs to be able to deal with mismatches and don't cares (corresponding to “assay failure”). The nature of the data allows a very efficient binary encoding with two bits per SNP. This yields a reasonable fast algorithm that just compares a pattern to each string in the set (the LS algorithm described and analyzed above). On the other hand, a trie can be used as an index for the data. Although a trie has the same worst-case look-up time, we can expect to save some work because fewer comparisons are necessary. As a drawback, the constants in the algorithm are a bit higher due to the tree structure involved. To choose the optimal algorithm, a more detailed analysis was needed, which we have presented in this chapter.

In the concrete case of the above application, the search is controlled by bounding the number of don't cares and the number of mismatches. We find that the average-case behavior, bounding only the don't cares, is approximately $O((\log n)^d n^{0.585})$ when allowing d don't cares. Bounding only the number of mismatches would result in Hamming distance, but in this application a don't care character cannot induce a mismatch. Therefore, the average-case complexity is approximately $O((\log n)^d n^{0.661})$ when allowing d mismatches. This is significantly worse than Hamming distance alone, which yields a search time of $O((\log n)^{d+1})$. It also dominates the bound on the number of don't cares. When deciding whether the LS or the TS algorithm should be used in this problem, we find that for $d > (3/8)\log_4 n - 1$ the LS algorithm will outperform the TS algorithm.

The results of this chapter can also be used to estimate the running time of the search methods described by Buchner et al. [BT03, BTG03]. They use a generalized suffix tree built on the strings representing the discretized angles of the three-dimensional structure of proteins. Searches are performed allowing some deviations. This is captured using the arithmetic distance defined in the previous section, a distance function that we can handle with our approach. In particular, the full range of 360 de-

grees is discretized into an alphabet $\Sigma = \{[0, 15), \ldots, [345, 360)\}$ of size twenty-four. The algorithm then searches a protein substructure by considering all angles within i intervals to the left and right, i.e., for $i = 2$ intervals to both sides, the interval $[0, 15)$ matches the intervals $[330, 345)$, $[345, 360)$, $[0, 15)$, $[15, 30)$, and $[30, 45)$. The probability of a match is thus $\frac{2i+1}{|\Sigma|}$. In their application, Buchner et al. [BT03, BTG03] allow no mismatch, i.e., the search is stopped if the angle is not within the specified range. The asymptotic running time can thus be estimated to be $O(n^{\log_{|\Sigma|}(2i+1)})$. Although a suffix tree is used, and we do not expect the underlying distribution of angles to be uniform and memoryless, this result can be used as a (rough) estimate, especially to understand the effect of different choices of i. The results in [AS92, JS94, JMS04], which point out a relation between the two trees at least for a memoryless source, support this claim.

Chapter 5

Text Indexing with Errors

In the last chapter, we saw how using a trie for the approximate text indexing problems $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{all}\rangle$ and $\langle \mathcal{P}(\Sigma^*)|\mathrm{d}|f(n)|\mathsf{doc}|\mathsf{pref}\rangle$ sped up searches over the naive method of comparing each string with the search pattern. The search time, however, depended on the size of the index, respectively the text corpus. In Chapter 2, we discussed the solution to some exact text indexing problems using (generalized) suffix trees. For this case, we were able to achieve a query time that was independent of the size of the text corpus. In fact, the algorithms were linear in the number of reported hits and the size of the search pattern, i.e., output sensitive. This is the best we can hope for in any text indexing problem. In this chapter, we present a data structure and corresponding algorithms for text indexing that are also output sensitive. This behavior comes at the cost of a larger index size. In the worst-case, the index may become very large, but we are able to bound the size on average and with high probability. The method can also be modified so that the size is bounded in the worst-case, although the algorithm is only output sensitive on average in this case.

We illustrate the basic idea for the approximate dictionary indexing problem allowing d mismatches. For a set S of cardinality n, the possible outputs for any query are the different strings matching a pattern w. For the empty string, we report all of S. When reading a pattern string from left to right, we can (possibly) exclude some strings from S for each character we have read because they cannot match the pattern with d or fewer mismatches. In particular, the prefix we have currently read from the pattern must match a prefix of a string with at most d mismatches.

To demonstrate the principle, assume that we perform the trie search of the previous chapter in a breadth first instead of a depth first manner. In the first round, we have not yet read anything, and the possible hits are all strings represented by the root of the tree. In a later round, we have read a prefix of length i of the pattern. The possible hits are represented by all subtrees of nodes at depth i that were reached with d or fewer mismatches. In the next round, we move to those children of the current nodes for that we still have at most d mismatches. Unless this applies to all children, they represent a subset of the strings of the previous step. We have eliminated from our current set those strings belonging to child nodes where we reached the limit of d mismatches.

The idea to achieve an optimal query time is to reduce the number of nodes actually visited to a constant number per examined pattern character. To this end, we need more

nodes to represent the different search states. Typically, we expect that after examining $O(\log n)$ characters of the pattern, there should not be more than a constant number of possible output candidates, which we can check in output optimal time. For pattern prefixes of length $c \log n$, there can be at most $|\Sigma|^{c \log n} = n^{c \log |\Sigma|}$ different states. We show that the number of states is even smaller on average, thus yielding an expected index size of $O(n \log^d n)$.

We elaborate the idea by distinguishing cases for the number of errors i from zero to d. For i errors, there is a threshold h_i for that no two strings in the set can be matched by the same string of length h_i with i errors. If the pattern is larger, there is at most one occurrence that we can check "manually". For shorter strings, we compute and store all possible string sets in a trie discounting the cases we already solved for smaller i. These tries are called error trees. The threshold h_i is the height of the i-th error tree.

At first glance, the idea may be reminiscent of the approach taken in [GMRS03] with the "repetition index" playing the role of our threshold. Conceptually, the main difference is that our approach is that of an index (or "register"), i.e., if the pattern is small, we can directly read the occurrences from the index structure, while the method described in [GMRS03] works more like a filter (or "accelerator") that reports areas of the strings with one or more matches. As a result, in contrast to [GMRS03], we can guarantee a worst-case look-up time.

The main text indexing data structure in [CGL04] is also based on error trees (called k-errata trees). In contrast to our approach, where all strings are merged into a single new tree, the trees in [CGL04] are recursively decomposed into centroid paths, and merging only takes place along these paths. A centroid path is a path that contains at each node the edge that leads to the subtree with the most leaves. Any path in a tree with n leaves can pass at most $\log n$ centroid paths. For each path, separate processing is done. On the one hand, this bounds the index size in the worst-case. On the other hand, this also requires to handle the paths separately during the search. Furthermore, it complicates the process of eliminating duplicate reports of occurrences.

The analysis of our error trees shows that they have a logarithmic height. This is used to bound the size of the trees on average. The intuition for this idea draws from the very detailed analysis of digital search trees. In particular, tries and suffix trees have an expected height of $O(\log n)$ (see, e.g., [Knu98, Pit85, KP86, JS91, Szp93a, Szp93b]). This height reflects the length of longest repeated substring in random sequences of length n. Even when allowing some errors, the length of such repeats is still $O(\log n)$ [ŁS97]. It is thus reasonable to suspect that error trees created by allowing a constant number of errors in prefixes up to the height of the suffix tree or trie behave similarly well and have a height of $O(\log n)$.

This chapter is organized as follows: After the formal problem definition, we take a closer look at the edit distance because it is inherently ambiguous. We then define a relaxed version of Σ^+-trees, which we need to realize a trade-off between time and space. Our basic indexing data structure is described next, followed by the search algorithm. Then the average size and the preprocessing for the data structure are analyzed. Finally, we present a trade-off between time and space, which allows to bound the size in the worst-case in exchange for a "weakening" of the query time bound from a worst-case to an average-case bound.

5.1 Definitions and Data Structures

Our indexing scheme can be applied to a variety of problems. The two basic parameters are a set S of strings, from which we build the index, and a criterion that we use to select leaves from subtrees.

Our basic data structures are representations of modified Σ^+-trees based on sets of strings. For any set of strings S, in the modified Σ^+-tree $\mathcal{T}$ corresponding to S, the number of leaves corresponds to the cardinality of S. The size of the data structure thus depends only on the cardinality of the set. By using different sets, we are able to solve different indexing problems. In all cases relevant here, the cardinality of the set is upper bounded by the size of any representation of the set. Therefore, as a unifying view in the description of the algorithm, we assume that we are given a set S of strings, called the *base set*. In particular, our data structure can be adapted to solve the following problems:

- For $\langle\Sigma^*|\mathsf{edit}|k|\mathsf{occ}|\mathsf{all}\rangle$ (approximate text indexing with occurrence reporting), the base set S is the set of suffixes of a string.
- For $\langle\Sigma^*|\mathsf{edit}|k|\mathsf{pos}|\mathsf{pref}\rangle$ (approximate text indexing with position reporting), the base set S is the set of suffixes of a string.
- For $\langle\mathcal{P}(\Sigma^*)|\mathsf{edit}|k|\mathsf{doc}|\mathsf{all}\rangle$ (approximate dictionary querying), the base set S is a set of independent strings.
- For $\langle\mathcal{P}(\Sigma^*)|\mathsf{edit}|k|\mathsf{doc}|\mathsf{pref}\rangle$ (approximate dictionary prefix querying), the base set S is a set of independent strings.
- For $\langle\mathcal{P}(\Sigma^*)|\mathsf{edit}|k|\mathsf{doc}|\mathsf{substr}\rangle$ (approximate document collection indexing with document reporting), the base set S is a set of suffixes of independent strings.
- For $\langle\mathcal{P}(\Sigma^*)|\mathsf{edit}|k|\mathsf{pos}|\mathsf{pref}\rangle$ (approximate document collection indexing with position reporting), the base set S is a set of suffixes of independent strings.
- For $\langle\mathcal{P}(\Sigma^*)|\mathsf{edit}|k|\mathsf{occ}|\mathsf{all}\rangle$ (approximate document collection indexing with occurrence reporting), the base set S is a set of suffixes of independent strings.

Before we can detail the selection process, we need some more definitions regarding the edit distance, the modified Σ^+-trees, and their representation. In the following, we assume that we are working with a set S of strings as specified above.

5.1.1 A Closer Look at the Edit Distance

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

Our indexing structure is based on the idea of computing certain strings that are within a specified distance to elements from our base set S or sets derived from S

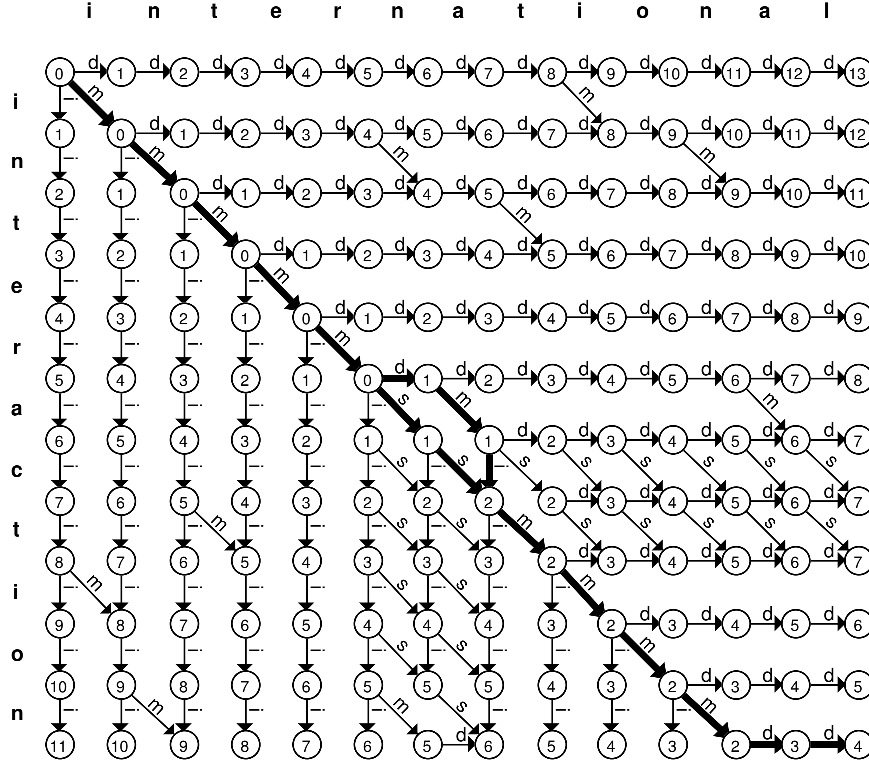


Figure 5.1: Example of a relevant edit graph $G^{\text{rel}}_{\texttt{international},\texttt{interaction}}$ for transforming the string `international` into the string `interaction`. Only relevant arcs are drawn, i.e., those arcs belonging to some shortest path. The edit paths between the two strings are marked in bold. An `s` indicates a substitution, a `d` a deletion, an `i` an insertion, and an `m` a match.

where the errors occur in prefixes of a bounded length. Therefore, we have to establish close ties between the length of minimal prefixes and the number of errors therein.

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

Definition 5.1 (k-Minimal prefix length).
For two strings $u, v \in \Sigma^*$ with $\mathrm{d}(u, v) = k$, we define

$$\begin{aligned}&\operatorname{minpref}_{k,u}(v)\\&\quad= \min\left\{l \;\middle|\; \mathrm{d}\left(u_{-,l}, v_{-,l+|v|-|u|}\right) = k \textit{ and } u_{l+1,-} = v_{l+|v|-|u|+1,-}\right\} \quad . \qquad (5.1)\end{aligned}$$

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

For a more detailed understanding of edit distance, it is helpful to consider the *edit graph*. The edit graph for the transformation of the string u into the string v contains a vertex[1] for each pair of prefixes (s, t) with $s \in \operatorname{prefixes}(u)$ and $t \in \operatorname{prefixes}(v)$. An

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

arc connects two vertices if the prefixes represented by the vertices differ by at most one character. Each arc is labeled by a weight corresponding to equation (2.9), i.e., the weight is zero if and only if the prefixes of the source vertex are both extended by the same character to form the prefixes of the target vertex (a match). Otherwise the weight of the arc is one, corresponding to a substitution (diagonal arcs), a deletion (horizontal arcs), or an insertion (vertical arcs). The vertex representing the empty prefixes $(\varepsilon, \varepsilon)$ is the start vertex and it is labeled with weight zero. Each vertex in the edit graph is labeled with the weight of a lightest (or shortest if we consider weights as distances) path connecting it to the start vertex. The vertex representing both strings (u, v) completely is the end vertex. Its label is the edit distance between both strings. We call an arc relevant if it lies on a shortest path from the start vertex to another vertex. The *relevant edit graph* contains only the relevant arcs. We denote the relevant edit graph for transforming u into v by $G^{\mathsf{rel}}_{u,v}$. An *edit path* is any path in the relevant edit graph connecting the start with the end vertex. Figure 5.1 shows an example. Each path connecting the start with the end vertex corresponds to a minimal set of edit operations to transform one string into the other.

Recall that in Section 2.1.3 the edit distance is defined as the minimal number of operations necessary to transform a string u into another string v. It is convenient to define the operators del, ins, and sub of type $\Sigma \cup \{\varepsilon\} \to \Sigma \cup \{\varepsilon\}$. If we have distance $\mathrm{d}(u, v) = k$ for two strings $u, v \in \Sigma^*$, then there exist one or more sequences of operations $(op_1, op_2, \ldots, op_k)$ with $op_i \in \{\mathrm{del}, \mathrm{ins}, \mathrm{sub}\}$ such that $v = op_k(op_{k-1}(\cdots op_1(u) \cdots))$. We call $u(i) = op_i(\cdots op_1(u) \cdots)$ the i-th *edit stage*. Each operation op_i in the sequence changes the current string at a position $\mathrm{pos}\,(op_i)$. An insertion changing $u = u_1 \cdots u_{i-1} u_i u_{i+1} \cdots u_m$ into $u_1 \cdots u_{i-1} a u_i \cdots u_m$ has position i, a deletion changing u into $u_1 \cdots u_{i-1} u_{i+1} \cdots u_m$ has position i, and a substitution changing u into $u_1 \cdots u_{i-1} a u_{i+1} \cdots u_m$ also has position i. We call a sequence $\rho(u, v) = (op_1, op_2, \ldots, op_k)$ of edit operations an *ordered edit sequence* if the operations are applied from left to right, that is, for op_i and op_{i+1}, we have $\mathrm{pos}\,(op_i) \leq \mathrm{pos}\,(op_{i+1})$ if op_i is a deletion, and $\mathrm{pos}\,(op_i) < \mathrm{pos}\,(op_{i+1})$ otherwise. There are two different paths to transform `international` into `interaction` with four operations in the example in Figure 5.1: either replace `n` by `a`, replace `a` by `c` and delete `a` and `l`; or delete `n`, insert `c`, and delete `a` and `l`. For each set of operations, there is also a number of different possibilities to create an edit sequence. There are two ordered edit sequences: $\mathrm{sub}_{6,\mathtt{a}}, \mathrm{sub}_{7,\mathtt{c}}, \mathrm{del}_{12}, \mathrm{del}_{12}$ and $\mathrm{del}_6, \mathrm{ins}_{7,\mathtt{c}}, \mathrm{del}_{12}, \mathrm{del}_{12}$. But the edit sequence $\mathrm{del}_{12}, \mathrm{del}_{12}, \mathrm{ins}_{8,\mathtt{c}}, \mathrm{del}_6$ also transforms `international` into `interaction`. Observe how changing the order of the operations also effects the positions in the string where the operations apply. For a fixed set of operations, there is a unique order (except for swapping identical del_i-operations). The charm of ordered edit sequences lies in the fact that they transform one string into another in a well-defined way.

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

Lemma 5.2 (One-to-one mapping between paths and edit sequences).
Let $u, v \in \Sigma^*$ be such that $\mathrm{d}(u, v) = k$. Each ordered edit sequence $\rho(u, v) =$

$(op_1, \ldots, op_k)$ corresponds uniquely to an edit path $\pi = (p_1, \ldots, p_m)$ in $G^{\mathsf{rel}}_{u,v}$.

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

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

Note that the position of the i-th operation op_i derived from the edit path was $\mathrm{pos}\left(op_i\right) = r + 1$, where r was the row number of the source vertex of the arc representing op_i in the first construction. Thus, if the operations have the positions $r_1 + 1, \ldots, r_k + 1$, then the row numbers of the source vertices of non-zero weight arcs are $r_1, \ldots, r_k$. In the second construction, we derived the existence of a vertex at the start of an arc representing the $(i + 1)$-th operation with row and column numbers

$r'-1$ and $c+r'-1-r$, where r' was the position of the $(i+1)$-th operation. Thus, if the non-zero weight arcs on the edit path start at the row numbers $r_1,\ldots,r_k$, then the operations have the positions $r_1+1,\ldots,r_k+1$. As a result, we have a one-to-one mapping. □

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

Lemma 5.3 (Edit stages of ordered edit sequences).
Let $u,v\in\Sigma^*$ be such that $\mathrm{d}(u,v)=k$. Let $\rho(u,v)=(op_1,\ldots,op_k)$ be an ordered edit sequence, and let $u(i)=op_i(\cdots op_1(u)\cdots)$ be the i-th edit stage. If we have $\mathrm{minpref}_{i,u(i)}(u)>h+1$, then there exists an $j>h$ with $v_{-,j}=u(i-1)_{-,j}$.

Proof. By Lemma 5.2, there is a unique path $\pi(u,v)$ in the relevant edit graph corresponding to the sequence $\rho(u,v)$. The same holds for any subsequence $\rho_i(u,u(i))=(op_1,\ldots,op_i)$. Because there are no more operations when transforming u into $u(i)$, the remaining path that is not identical with the path for $\rho(u,v)$ must be made up of zero-weight arcs. Let p be the source vertex of the arc for the i-th operation on the edit path in the relevant edit graph $G^{\mathsf{rel}}_{u,u(i)}$. The vertex p must have weight $i-1$. Let q be the target vertex of the arc for the $(i-1)$-th operation. Up to vertex q, the edit paths in the relevant edit graphs $G^{\mathsf{rel}}_{u,u(i)}$ and $G^{\mathsf{rel}}_{u,u(i-1)}$ are equal. Thus, they are equal up to vertex p because there are only zero-weight arcs between q and p in the relevant edit graphs transforming u into $u(i)$. Furthermore, the same path is also found in the relevant edit graph $G^{\mathsf{rel}}_{u,v}$. Let the row and column numbers of p be r and c. Thus, p represents the prefixes $u_{-,c}$ and

$$v_{-,r}=u(i)_{-,r}=u(i-1)_{-,r}\ . \tag{5.2}$$

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

$$h+1<\mathrm{minpref}_{i,u(i)}(u)\le r'\le r+1\ , \tag{5.3}$$

and thus we have $r>h$. Equations (5.2) and (5.3) prove our claim. □

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

5.1.2 Weak Tries

We basically assume that all Σ^+-trees are represented by compact tries or by the similar weak tries. Therefore, we consider the strings in underlying sets to be independent and make no use of any intrinsic relations among them. A suffix tree for the string t

can also be regarded as a trie for the set $S = \text{suffixes}(t)$, which can be represented in size $O(|t|)$. The major advantage of the suffix tree is that (by using the properties of S) it can be built in time $O(|t|)$, whereas building a trie for $\text{suffixes}(t)$ may take time $O(l|t|)$, where l is the length of the longest common repeated substring in t. For our index data structure, this additional cost will only be marginal. Therefore, for the construction, we just regard all Σ^+-trees as compact tries and make no use of internal relations among the strings. In the following, let $\mathcal{T}(S)$ denote the compact trie for the string set $S \subset \Sigma^+$.

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

Definition 5.4 (l-Weak Σ^+-tree).
For $l > 0$, an l-weak Σ^+-tree $\mathcal{T}$ is a rooted, directed tree with edge labels from Σ^+. For every node p with $\text{depth}(p) < l$, there is at most one outgoing edge for each $a \in \Sigma$ whose label starts with the character a (relaxed unique branching criterion).

Although there is no one-to-one correspondence between strings and nodes in the l-weak Σ^+-tree, the definitions for the path of a node, the depth of a node, and the word set of the tree as given in Section 2.1.2 apply.

A *weak trie* is the representation of a compact weak Σ^+-tree for a set of strings S.[2] Indeed, as argued in Section 2.1.2, if we use virtual nodes, it makes no conceptual difference whether we compactify our trees or not. Thus, we will not be too rigid on this subject where it is not needed.

Definition 5.5 (l-Weak trie).
For a set of strings S, the l-weak trie $\mathcal{W}_l(S)$ is a compact implementation of an l-weak Σ^+-tree such that

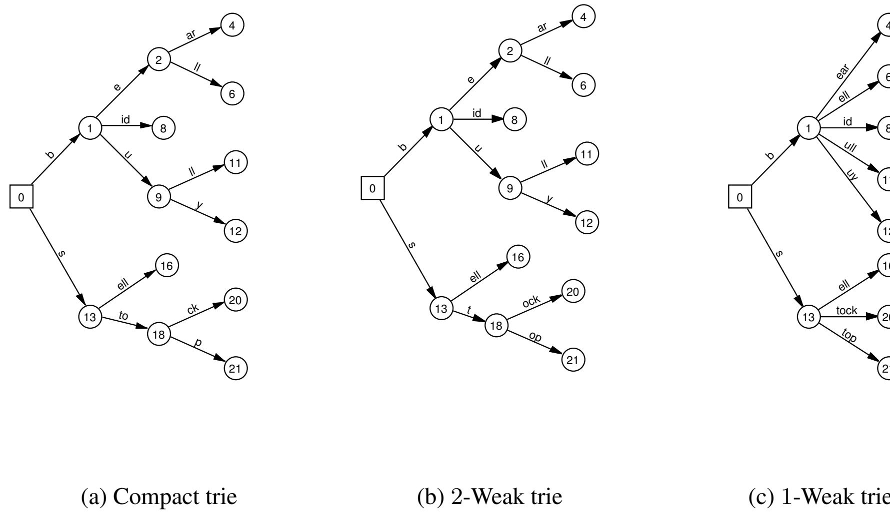
- *it contains exactly one leaf for each string in S, and*
- *there is either a single branching node at depth l, under which all leaves are attached, or there is only a single leaf and no branching after depth l.*

In any case, the largest depth of a branching node in an l-weak trie is l. Figure 5.2 shows examples of weak tries derived from the examples in Section 2.3.1. The height of the trie in Figure 5.2(a) is three, thus the 3-weak trie is the same as the compact trie. Because $\text{maxpref}(S)$ is the height of the trie for the string set S, the weak trie $\mathcal{W}_{\text{maxpref}(S)}(S)$ is just the compact trie $\mathcal{T}(S)$.

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

Definition 5.5 guarantees that weak tries are unique with respect to a string set S: Because the trie is compact (by the unique branching criterion), the nodes up to depth l are uniquely defined by the prefixes of length l of the strings in S. Each element in S has a prefix of length l that uniquely corresponds to a path of length l. If there is more than one string with a certain prefix u, then these are all represented by leaves under a node p with $\text{path}(p) = u$.

[2] The term "weak compact trie" would be more exact, but we prefer to use the shorter term.



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

Figure 5.2: Examples of a compact trie, a 1-weak trie, and a 2-weak trie for the prefix-free set of strings `bear`, `bell`, `bid`, `bull`, `buy`, `sell`, `stock`, `stop`.

5.2 Main Indexing Data Structure

In order to answer a query for any of our indexing problems, we have to find prefixes of strings in S that match the search pattern w with at most k errors. Assume that $S \subset \Sigma^*$ and $w \in \Sigma^*$ are given. We call $s \in S$ a *k-error length-l occurrence of w* if $\mathrm{d}\,(s_{-,l}, w) = k$. We then also say that w matches s up to length l with k errors. We will omit the length l if it is either clear from the context or irrelevant. To solve the indexing problems defined in Section 5.1 in optimal time, we need to be able to find all d-error occurrences of the search pattern w in time $O(|w|)$. For that purpose, we will define error trees so that leaves in certain subtrees contain the possible matches. From these, we select the desired output using range queries.

On an abstract level, the basic idea of the index for d errors is the following: A single string $s \in S$ can be compared to the pattern w to check whether w matches a prefix of s with at most d errors in time $O(d|w|)$ as described in Section 5.1.1. On the other hand, a precomputed index of all strings within edit distance d of S (e.g., a trie containing all strings r for which r matches a prefix of a string $s \in S$ with at most d errors) allows a linear look-up time. Both methods are relatively useless by themselves: The index would be prohibitively large, whereas comparing the pattern to each string would take too long. Therefore, we take a balanced approach by precomputing some of the neighboring strings and directly computing the distance for others. In particular, we inductively define sets of strings $W_0, \ldots, W_d$, where, for $0 \leq i \leq d$, the set W_i consists of strings with edit distance i to some string in S. We begin with $S = W_0$. Parameters $h_0, \ldots, h_d$ are used to control the precomputing. As an example, W_1 consists of all strings $r = u'v$ such that $\mathrm{d}\,(r, s) = 1$ for some $s = uv \in W_0$ and $|u'| \leq h_0 + 1$, i.e., we apply all possible edit operations to strings $s \in S$ that result

in a modified prefix of length $h_0 + 1$. The partitioning into $d + 1$ sets allows searching with different error bounds up to d.

5.2.1 Intuition

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

We extend this scheme to two errors as follows. Assume that w matches a prefix $s \in \text{prefixes}(S)$ of a string in the base set with two errors. There are three cases depending on the positions of the errors (of an ordered edit script). If the first error occurs above the height h_0 of the trie $\mathcal{T}$, then we find a prefix of w in $\mathcal{T}$ leading to a leaf edge. Again, we can check the single string corresponding to this edge in time $O(|w|)$ as above. If the first error occurs below height $h_0 + 1$, then we have inserted a string r into the trie $\mathcal{T}'$ that reflects this first error. We are left with two cases: Either the second error occurs above the height h_1 of $\mathcal{T}'$ or below. In the first case, there is a leaf representing a single string[4], which we can check in time $O(|w|)$. Finally, for the case that the second error occurs below h_1, we generate $O(h_1)$ new strings for each string in $\mathcal{T}'$ in the same manner as before. The new strings are inserted in a trie $\mathcal{T}''$ with height h_2. Again, we find all matches of w in the subtree $\mathcal{T}''_w$.

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

5.2.2 Definition of the Basic Data Structure

We now rigidly lay out the idea. There are some more obstacles that we have to overcome when generating new strings from a set W_i: We have to avoid generating a string by doing a reverse error, i.e., undoing an error from a previous step. In addition, to ease representation, we allow all errors on prefixes of strings that lead to a new prefix of maximal length $h_{i-1} + 1$. In particular, this also captures all operations with positions up to $h_{i-1} + 1$. (See the proof of Lemma 5.2: The position of an operation corresponds to one plus the row number of the source vertex in the edit graph. The resulting string has length at most one plus this row number.) The key property of the following inductively defined sets $W_0, \ldots, W_d$ is that we have at least one error before

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

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

$h_0 + 1$, two errors before $h_1 + 1$, and so on until we have d errors before $h_{d-1} + 1$ in W_d.

Definition 5.6 (Error sets).
For $0 \le i \le d$, the error set W_i is defined inductively. The first error set W_0 is defined by $W_0 = S$. The other sets are defined by the recursive equation

$$W_i = \Gamma_{h_{i-1}}(W_{i-1}) \cap S_i \ , \tag{5.4}$$

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

$$\Gamma_l = \{op(u)v \mid uv \in A, |op(u)| \le l + 1, op \in \{\mathrm{del}, \mathrm{ins}, \mathrm{sub}\}\} \ , \tag{5.5}$$

the set S_i is defined as

$$S_i = \{r \mid \textit{there exists } s \in S \textit{ such that } \mathrm{d}(s, r) = i\} \tag{5.6}$$

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

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

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

Lemma 5.7 (Error positions in error sets).
Assume that the string $r \in W_i$ stems from the string $u \in S$ by a sequence of operations $op_1, \ldots, op_i$. If the parameters h_i are strictly increasing for $0 \le i \le d$, then $r_{h_{i-1}+2,-} = u_{h_{i-1}+2,-}$ and $r_{h_i+1,-} = u_{h_i+1,-}$.

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

Thus, after $i - j$ operations, $p_j \leq \operatorname{pos}(op_j) + i - j$. By Definition 5.6, we have $\operatorname{pos}(op_j) \leq h_{j-1} + 1$. Therefore, in step i, we have $p_j \leq h_{j-1} + 1 + i - j$. Because the h_i are strictly increasing, we have $h_{j-1} + 1 \leq h_j \leq h_{j+1} - 1 \leq \cdots \leq h_{i-1} + 1 - i + j \leq h_i - i + j$. As a result, for any position in r, where a character was changed, we have $p_j \leq h_{i-1} + 1 \leq h_i$. □

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

Definition 5.8 (Error trees).
The i-th error tree $\mathrm{et}_i(S)$ is defined as the weak trie $\mathcal{W}_{h_i}(W_i)$. Each leaf p of $\mathrm{et}_i(S)$ is labeled with (id_s, l), for each string $s \in S$, where id_s is an identifier for s and $l = \operatorname{minpref}_{i,\operatorname{path}(p)}(s)$.

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

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

5.2.3 Construction and Size

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

Lemma 5.9 (Construction and size of W_i and $\mathrm{et}_i(S)$).
For $0 \leq i \leq d$, assume that the parameters h_i are strictly increasing, i.e., $h_i > h_{i-1}$, and let $n_i = |W_{i-1}|$. The set W_i can be constructed from the set W_{i-1} in time $O(n_{i-1}h_{i-1}h_i)$ and space $O(n_{i-1}h_{i-1})$ yielding the error tree $\mathrm{et}_i(S)$ as a byproduct. The i-th error tree $\mathrm{et}_i(S)$ has size $O(|S|h_0 \cdots h_{i-1})$.

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

Let W_i' be the multi-set of all strings constructed from W_{i-1} by applying all possible edit operations that result in modified prefixes of length $h_{i-1}+1$. We have to avoid

undoing any earlier operation. This can be accomplished by comparing the new operation with the annotated edit sequence. Note that we may construct the same string from multiple edit sequences (also including the ordered edit sequence). To construct W_i from W'_i, we have to eliminate the duplicates.

By Lemma 5.7, if the string $r \in W'_i$ stems from the string $u \in S$, then we have $r_{h_i+1,-} = u_{h_i+1,-}$. If $r, s \in W'_i$ are equal, then they must either both stem from the same string $u \in S$, or they are both suffixes of the same string u. Because there are no errors after h_i, we have $r_{h_i+1,-} = s_{h_i+1,-} = u_{h_i+1,-}$. Note that $h_i + 1 - i \leq |u_{h_i+1,-}| \leq h_i + 1 + i$, so there can be at most $2i+1$ different suffixes for any string u.

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

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

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

We choose the parameters h_i either as $h_i = \mathrm{maxpref}(W_i)$ or as $h_i = h + i$ for some fixed value h that we specify later. By both choices, we satisfy the assumptions of Lemma 5.9 as shown by the following lemma.

Lemma 5.10 (Increasing common prefix of error sets).
For $0 \leq i \leq d$, let $h_i = \mathrm{maxpref}(W_i)$ be the maximal prefix of any two strings in the i-th error set, then we have $h_i > h_{i-1}$.

Proof. We prove by induction. Let r and s be two strings in W_{i-1} with a maximal prefix $r_{-,h_{i-1}} = s_{-,h_{i-1}} = u$ for some $u \in \Sigma^{h_{i-1}}$. Because r and s are not identical, we have $r = uav$ and $s = ubv'$ for some strings $v, v' \in \Sigma^*$ and $a, b \in \Sigma$. We have $|\Sigma| \leq 2$; therefore, $\Gamma(W_{i-1})$ contains at least the strings ubv, $uaav$, $ubav$, uv, uav', $uabv'$, $ubbv'$, and uv'. By Lemma 5.7, for any string $r \in W_{i-1}$ that stems from $t \in S$, we have $r_{h_{i-1}+1,-} = t_{h_{i-1}+1,-}$, thus no character at a position greater or equal to $h_{i-1} + 1$ can have been changed by a previous operation. Hence, applying any operation at $h_{i-1} + 1$ creates a new string that has distance i to some string in S. Both $ubav$ and $ubbv'$ were created by inserting a character at $h_{i-1} + 1$, so they do not undo an operation and belong to W_i. They both have a common prefix of length at least $h_{i-1} + 1 > h_{i-1}$. Therefore, we find that $h_i > h_{i-1}$. ☐

For $h_i = \mathrm{maxpref}(W_i)$, we do not need to use the buckets as described in the proof of Lemma 5.9, but we can create the error trees more easily. By assumption, two strings are either completely identical, or they have a common prefix of size at most h_i. On the other hand, no two strings have a common prefix of more than h_i, thus there can be at most one bucket below h_i. If two strings r and s are identical, they must stem from suffixes of the same string $u \in S$ and the suffixes $r_{h_{i+1},-}$ and $s_{h_{i+1},-}$ must

be equal. Because s and r are implemented by pointers to strings in S, we can easily check whether they reference the same suffix of the same string during the process of insertion and before comparing at a character level.

5.2.4 Main Properties

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

Lemma 5.11 (Existence of matches).
Let $w \in \Sigma^*$, $s \in S$, and $t \in \text{prefixes}(s)$ be such that $\mathrm{d}(w,t) = i$. Let $\rho(w,t)$ be an ordered edit sequence for $w = op_i(\cdots op_1(t)\cdots)$. For $0 \leq j \leq i$, let $t(j) = op_j(\cdots op_1(t)\cdots)$ be the j-th edit stage. If for all $1 \leq j \leq i$ we have

$$\text{minpref}_{j,t(j)}(t) \leq h_{j-1} + 1 \ , \tag{5.7}$$

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

$$w \in \text{prefixes}(r), \quad r = op_i(\cdots op_1(s)\cdots), \quad \textit{and} \quad l = \text{minpref}_{i,r}(s) \ . \tag{5.8}$$

Proof. We prove by induction on i. For $i = 0$, we have $W_0 = S$ and the claim is obviously true because $w = t \in \text{prefixes}(s)$ in that case.

Assume the claim is true for all $j \leq i-1$. Let $r = op_i(\cdots op_1(s)\cdots)$. Because $\mathrm{d}(r,s) = i$, we have to show that there exists a string $r' \in W_{i-1}$ and strings $u, u', v \in \Sigma^*$, with $r = uv$, $r' = u'v$, $\mathrm{d}(u,u') = 1$, and $|u'| \leq h_{i-1}+1$. Then we have $r \in W_i$ by Definition 5.6.

Set $l = \text{minpref}_{i,r}(s)$. Because w and t are prefixes of r and s, which already have distance i, we have $l = \text{minpref}_{i,w}(t)$ and $l \leq h_{i-1}+1$ by equation (5.7). Set $u = s_{-,l+|s|-|r|}$, $u' = r_{-,l}$, and $v = r_{l+1,-}$. By Definition 5.1, we have $r = u'v$, $s = uv$, $\mathrm{d}(u,u') = i$, and $u' = op_i(\cdots op_1(u)\cdots)$. Set $u'' = op_{i-1}(\cdots op_1(u)\cdots)$ and $r' = u''v$, then we get $r' = op_{i-1}(\cdots op_1(s)\cdots)$. We are finished if we can show that $r' \in W_{i-1}$ because we have $|u'| = l \leq h_{i-1}+1$ and $\mathrm{d}(u',u'') = 1$.

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

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

Lemma 5.12 (Occurrences leading to matches).
For $r \in W_i$, let $w \in \text{prefixes}(r)$ be some prefix of r, and let $s \in S$ be a string corresponding to r such that $l = \text{minpref}_{i,r}(s)$. There exists a prefix $t \in \text{prefixes}(s)$ such that $\mathrm{d}(t,w) = i$ and $r_{|w|+1,-} = s_{|t|+1,-}$ if and only if $|w| \geq l$.

Proof. If $|w| \geq l$, then there are strings $u, v \in \Sigma^*$ with $w = uv$ and $u = w_{-,l}$. Because w is a prefix of r, we have $r = wx = uvx$. By Definition 5.1, there also exists a prefix $u' \in \text{prefixes}(s)$ with $s = u'vx$ and $\mathrm{d}(u', u) = i$. Hence, we have $t = u'v \in \text{prefixes}(s)$, $\mathrm{d}(w, t) = \mathrm{d}(uv, u'v) = i$, and $x = r_{|w|+1,-} = s_{|t|+1,-}$.

Conversely, assume that $|w| < l$ and that there exists a prefix $t \in \text{prefixes}(s)$ with $\mathrm{d}(t, w) = i$ and $r_{|w|+1,-} = s_{|t|+1,-}$. Set $m = |w|$ and recall that $w = r_{-,m}$. Then $s = tr_{|w|+1,-}$ and, thus we have $t = s_{-,|s|-|r|+m}$. Therefore, we find that $\mathrm{d}(r_{-,m}, s_{-,m+|s|-|r|}) = i$ and $r_{m+1,-} = s_{m+1+|s|-|r|,-}$, i.e., m is a candidate for $\mathrm{minpref}_{i,r}(s)$, which is a contradiction to $m < l$. □

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

Lemma 5.13 (Occurrence properties).
Assume w matches $t \in \text{prefixes}(s)$ for $s \in S$ with i errors, i.e., $\mathrm{d}(w, t) = i$. Let $\rho = (op_1, \ldots, op_i)$ be an ordered edit sequence such that $w = op_i(op_{i-1}(\cdots op_1(t) \cdots))$. There are two mutually exclusive cases,

(1) either $w \in \text{prefixes}(W_i)$, or

(2) there exists at least one $0 \leq j \leq i$ for that we find a string $r \in W_j$ such that $r = op_j(\cdots op_1(s) \cdots)$, and we have $w' = w_{-,l}$ for some $w' \in \text{prefixes}(r)$ and some $l > h_j$.

Proof. Let $t(j) = op_j(\cdots op_1(t) \cdots)$ be the j-th edit stage. Assume that there exists no j such that $\mathrm{minpref}_{j,t(j)}(t) > h_{j-1} + 1$. Then $w \in \text{prefixes}(W_i)$ by Lemma 5.11. Otherwise, let j be the smallest index such that $\mathrm{minpref}_{j+1,t(j+1)}(t) > h_j + 1$. By Lemma 5.11, there exists a string $r \in W_j$ with $t(j) \in \text{prefixes}(r)$. By Lemma 5.3, there exists a prefix $w' = w_{-,l} = t(j)_{-,l}$ with $l > h_j$, thus $w' \in \text{prefixes}(r)$. □

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

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

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

5.2.5 Search Algorithms

Lemmas 5.12 and 5.13 directly imply a search algorithm along the case distinction made above. Recall that we can build the index efficiently if we choose the parameters h_i either as $h_i = \mathrm{maxpref}(W_i)$ or as $h_i = h + i$ for some fixed value h. The index supports searches if either $h_i = \mathrm{maxpref}(W_i)$ or the length of the search pattern w is bounded by $|w| \leq h$. For these parameters, we can check all prefixes $t \in \text{prefixes}(S)$ for which Case B applies in time $O(|w|)$: If $|w| \leq h$, then we can never have $|w'| > h_i$ for a prefix $w' \in \text{prefixes}(w)$ and so the case never applies.

Otherwise we have $h_i = \mathrm{maxpref}(W_i)$. In this case, the error trees become tries and so there is at most one leaf in $\mathrm{et}_i(S)_{w'}$ if the length of the prefix $w' \in \mathrm{prefixes}(w)$ is greater than h_i. Each leaf can have at most $2d + 1$ labels corresponding to at most $2d+1$ strings from S. We compute the edit distance of w to every prefix of each string in time $O(|w|)$ with a d-bounded computation of the edit distance (see Section 5.1.1). There can be at most $2d + 1$ prefixes that match w, thus from all $d + 1$ error trees, we have at most $d(2d + 1)^2$ strings in total for which we must check the problem specific conditions for reporting them. As a result, we get the following lemma.

Lemma 5.14 (Search time for Case B).
If d is constant and either $h_i = \mathrm{maxpref}(W_i)$ or the length of the search pattern w is bounded by $|w| \leq h \leq \min_i h_i$, then the total search time spent for Case B is $O(|w|)$.

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

The strings in the base set S can be either independent or they are suffixes of a string u (also called document). Recalling the definition for the base set made in Section 5.1, we have to support four different **types** of selections:

1. For the following problems, we want to report each prefix t of any string $s \in S$ that matches the pattern w with at most d errors:
   - $\langle \Sigma^* | \mathsf{edit} | k | \mathsf{occ} | \mathsf{all} \rangle$
   - $\langle \mathcal{P}(\Sigma^*) | \mathsf{edit} | k | \mathsf{occ} | \mathsf{all} \rangle$
2. For the following problems, we want to report each string $s \in S$ for which a prefix $t \in \mathrm{prefixes}(s)$ matches the pattern w with at most d errors:
   - $\langle \Sigma^* | \mathsf{edit} | k | \mathsf{pos} | \mathsf{pref} \rangle$
   - $\langle \mathcal{P}(\Sigma^*) | \mathsf{edit} | k | \mathsf{doc} | \mathsf{pref} \rangle$
   - $\langle \mathcal{P}(\Sigma^*) | \mathsf{edit} | k | \mathsf{pos} | \mathsf{pref} \rangle$
3. For the following problems, we want to report each string $s \in S$ which matches the pattern w with at most d errors:
   - $\langle \mathcal{P}(\Sigma^*) | \mathsf{edit} | k | \mathsf{doc} | \mathsf{all} \rangle$
4. For the following problems, we want to report each document u if a prefix $t \in \mathrm{prefixes}(s)$ of a suffix $s \in S$ of u matches the pattern w with at most d errors:
   - $\langle \mathcal{P}(\Sigma^*) | \mathsf{edit} | k | \mathsf{doc} | \mathsf{substr} \rangle$

The basic task is to match the pattern in each of the $d + 1$ error trees. If the complete pattern could be matched, we are in Case A. To support the selection of the different types, we create additional arrays for each error tree. Let n_i be the number

of leaf labels in the i-th error tree $\mathrm{et}_i(S)$. First, we create an array A_i of size n_i that contains each leaf label and a pointer to its leaf in the order encountered by a depth first traversal of the error tree. For example, if the weak tree in Figure 5.2(c) were an error tree, we would first store the labels of the node 4, then all labels of the node 6, and so on until we would have stored all labels of the node 21 at the end of A_i. The order of the depth first traversal can be arbitrary but must be fixed. Each node p of the error tree is annotated by the leftmost index $\mathrm{left}(p)$ and the rightmost index $\mathrm{right}(p)$ in A_i containing a leaf label from the subtree rooted under p. For a virtual node q, $\mathrm{left}(q)$ and $\mathrm{right}(q)$ are taken from the next non-virtual node below q.

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

By Lemma 5.12, for reports of Type 1, we have to select the strings corresponding to those labels for that the minimal prefix value l stored in the label is smaller than the length of the pattern. This is achieved by setting $B_i[j]$ to l if $A_i[j]$ contains the leaf label (id_s, l). Let p be the (virtual) node corresponding to the location of the pattern w in the error tree $\mathrm{et}_i(S)$. A bounded value range (BVR) query $(\mathrm{left}(p), \mathrm{right}(p))$ with bound $|w|$ on B_i yields the indices of labels in A_i for that $l \leq |w|$ in linear time in the number of labels.

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

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

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

Lemma 5.15 (Search time for Case A).
If d is constant and either $h_i = \mathrm{maxpref}(W_i)$, or the length of the search pattern w is bounded by $|w| \leq h \leq \min_i h_i$, then the total search time spent for Case A is $O(|w| + \mathrm{occ})$, where occ is the number of reported outputs.

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

For the space usage, we have the following obvious lemma.

Lemma 5.16 (Additional preprocessing time and space for Case A).
The additional space and time needed for the range queries and the arrays for solving Case A is linear in the number of leaves of the errors trees.

Proof. There are at most $2d + 1$ labels per leaf, and so the size of the generated arrays is linear in the number of leaves. The time needed for the depth first traversals is also linear in the array sizes. Finally, the range queries are also prepared in time and space linear in the size of the arrays (see Section 2.3.2). □

5.3 Worst-Case Optimal Search-Time

When setting h_i to $\mathrm{maxpref}(W_i)$, our main indexing data structure already yields worst-case optimal search-time by Lemmas 5.14 and 5.15. What is left is to determine the size of the data structure and the time needed for preprocessing. Note that, if S is the set of suffixes of a string of length n, then already $h_0 = \mathrm{maxpref}(W_0) = \mathrm{maxpref}(S)$ can be of size $\Omega(n)$. For independent strings, the worst-case size of $h_0 = \mathrm{maxpref}(S)$ cannot be bounded at all in terms of $n = |S|$. Fortunately, the average size is much better and it occurs with high probability. In this section, we derive the corresponding average-case bounds for $h_i = \mathrm{maxpref}(W_i)$. Together with Lemmas 5.9 and 5.16, this gives a bound on the total size and preprocessing time because they are all dominated by the size and preprocessing time needed for the d-th error tree:

Corollary 5.17 (Data structure size and preprocessing time).
Let n be the number of strings in the base set S. For constant d, the total size of the main data structures is $O(nh_0h_1 \cdots h_{d-1})$, and the time needed for preprocessing is $O(nh_0h_1 \cdots h_{d-1}h_d)$.

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

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

Lemma 5.18 (Length of common or repeated substrings).
Let $W_0, \ldots, W_d$ be the error sets by Definition 5.6. If there exists an i with $h_i \geq (2i + 1)l$ for $l > 1$, then there exists a string v of length $|v| > l$ such that either v is a substring of two independent strings in S, or $v = u_{j,j+l-1} = u_{j',j'+l-1}$ with $j \neq j'$ for some $u \in S$. Furthermore, for $l \geq 2$, both occurrences of v start in a prefix of length $2il$ of strings in S.

Proof. We prove the claim by induction over i. For $i = 0$, the claim is naturally true because h_0 is the length of the longest prefix between two strings in S. This is either

the longest repeated substring in a string u (if all suffixes of u were inserted into S), or the longest common prefix of two independent strings.

For the induction step, assume that we have $h_j < (2j+1)l$ for all $j < i$ and that $h_i \geq (2i+1)l$. Let $r, s \in W_i$ be two strings with a common prefix v of length $|v| = h_i \geq (2i+1)l$, thus $v = r_{-,|v|} = s_{-,|v|}$. Let $t^{(r)}, t^{(s)} \in S$ be the elements from the base set corresponding to r and s, i.e., $\mathrm{d}\,(t^{(r)}, r) = i$ and $\mathrm{d}\,(t^{(s)}, s) = i$. Recall that, by Lemmas 5.7 and 5.10, $r_{h_{i-1}+2,-} = t_{h_{i-1}+2,-}$ if $r \in W_i$ stems from $t \in S$. It follows that $t^{(r)}$ and $t^{(s)}$ share the same substring $w = {t^{(r)}}_{h_{i-1}+2,h_i} = {t^{(s)}}_{h_{i-1}+2,h_i}$ of length $|w| = h_i - h_{i-1} - 2 + 1 > (2i+1)l - (2(i-1)+1)l - 1 = 2l - 1$. Even if $t^{(r)}$ and $t^{(s)}$ are the same string u or suffixes of the same string u, then w cannot start at the same position in u: Assume for contradiction that $w = {t^{(r)}}_{h_{i-1}+2,h_i} = {t^{(s)}}_{h_{i-1}+2,h_i} = u_{k,k+|w|}$, then we have ${t^{(r)}}_{h_{i-1}+2,-} = u_{k,-} = {t^{(s)}}_{h_{i-1}+2,-}$, so r and s would not branch at h_i, which would be a contradiction.

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

For the analysis, we assume the mixing model introduced in Section 2.2.3. The intuition for the next theorem is as follows. The height of (compact) tries and suffix trees is bounded by $O(\log n)$, where n is the cardinality of the input set for tries or the size of the string for suffix trees (see, e.g., [AS92] or [Szp00]). When allowing an error on prefixes bounded by the height, we essentially rejoin some strings that were already branching. The same process can take place again with the rejoined strings. Thus, the height of the i-th error tree should behave no worse than i times the height of the trie or the suffix tree. Although this bound may not be very tight, we prove exactly along this intuition. We conjecture that the heights actually behave much better.

Theorem 5.19 (Average data structure size and preprocessing time).
Let n be the number of strings in the base set S, which contains strings and suffixes of strings generated independently and identically distributed at random by a stationary ergodic source satisfying the mixing condition. For any constant d, the average total size of the data structures is $O(n \log^d n)$, and the average time for preprocessing is $O(n \log^{d+1} n)$. Furthermore, these complexities are achieved with high probability $1 - o(n^{-\epsilon})$ (for some $\epsilon > 0$).

Proof. By Lemma 5.18, if there exists an i such that $h_i \geq (2i+1)l$, then we find a string v of length $|v| \geq l$ that is a repeated substring of an independent string or a common substring of two independent strings. Let h^{rep} be the maximal length of any repeated substring in a single independent string in S, let h^{suf} be the maximal length of any repeated substring of a string u for that we have inserted all suffixes into S, and let h^{com} be the maximal length of any common substring of two independent strings. If we bound h^{rep}, h^{suf}, and h^{com} by l, we also bound h_i by $(2i+1)l$. We first turn to long substrings common to independent strings.

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

by equation (2.14). By Lemma 5.18, the common substrings must be found in prefixes of length $2il$ of a string in S. As a result, we find

$$\begin{aligned}\Pr\left\{h^{\mathsf{com}} \geq l\right\} &\leq \Pr\left\{\bigcup_{r,s\in S, 1\leq i,j\leq 2il}\{C_{i,j}\geq l\}\right\} \\ &\leq \sum_{r,s\in S,1\leq i,j\leq 2il} \Pr\left\{C_{i,j}\geq l\right\} = \sum_{r,s\in S,1\leq i,j\leq 2il}\mathbb{E}\left[w^l\right] \\ &\leq 4n^2l^2i^2\mathbb{E}\left[w^l\right] \leq cn^2l^2i^2e^{-2r_2 l} \quad (5.9)\end{aligned}$$

for some constant c and growing l.

For the maximal lengths h^{rep} and h^{suf} of repeated substrings, we use known results from [Szp93a], where the length $h^{(m)}$ of a repeated substring in a string of length m is bounded by

$$\Pr\left\{h^{(m)} \geq l\right\} \leq c'm\left(l\sqrt{\mathbb{E}\left[w^l\right]} + m\mathbb{E}\left[w^l\right]\right) \leq cmle^{-r_2 l} \quad (5.10)$$

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

$$\Pr\left\{h^{\mathsf{rep}} \geq l\right\} \leq cil^2e^{-r_2 l} \quad , \quad (5.11)$$

and

$$\Pr\left\{h^{\mathsf{suf}} \geq l\right\} \leq cnle^{-r_2 l} \quad . \quad (5.12)$$

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

$$\begin{aligned}\Pr\left\{h_i \geq l\right\} &\leq \Pr\left\{\max\{h^{\mathsf{rep}}, h^{\mathsf{suf}}, h^{\mathsf{com}}\} \geq \frac{l}{2i+1}\right\} \\ &\leq \Pr\left\{h^{\mathsf{rep}} \geq \frac{l}{2i+1}\right\} + \Pr\left\{h^{\mathsf{suf}} \geq \frac{l}{2i+1}\right\} + \Pr\left\{h^{\mathsf{com}} \geq \frac{l}{2i+1}\right\} \\ &\leq c_1\frac{il^2}{(2i+1)^2}e^{-r_2\frac{l}{2i+1}} + c_2 n\frac{l}{2i+1}e^{-r_2\frac{l}{2i+1}} + c_3n^2l^2i^2e^{-2r_2\frac{l}{2i+1}} \\ &\leq cn^2l^2i^2e^{-r_2\frac{l}{2i+1}} \quad , \quad (5.13)\end{aligned}$$

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

$$\Pr\left\{h_i \geq (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}\right\} \leq c(1+\epsilon)^2 i^4 \ln^2 n^{-2\epsilon} n \quad , \quad (5.14)$$

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

The expected size can be bounded by

$$\begin{aligned}
\mathbb{E}\left[h_i\right] &\leq (1 - o(n^{-\epsilon}))c\log n + c' \sum_{l \geq (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}} n^2 l^3 i^2 e^{-r_2 \frac{l}{2i+1}} \\
&\leq c\log n + c' \sum_{l\geq 0} n^{-2\epsilon}\left(l + (1+\epsilon)2(2i+1)\frac{\ln n}{r_2}\right)^3 i^2 e^{-r_2\frac{l}{2i+1}} \\
&\leq c\log n + c'' i^5 n^{-2\epsilon}\ln^3 n \sum_{l\geq 0} l^3 e^{-r_2\frac{l}{2i+1}} \\
&= O\left(\log n\right) \quad (5.15)
\end{aligned}$$

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

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

5.4 Bounded Preprocessing Time and Space

In the previous section, we achieved a worst-case guarantee for the search time. In this section, we describe how to bound the index size in the worst-case in trade-off to having an average-case look-up time. Therefore, we fix h_i in Definitions 5.6 and 5.8 to $h_i = h + i$ for some h to be chosen later (but the same for all error trees). By Lemmas 5.9, 5.16, and 5.18, the size and preprocessing time is $O(nh^d)$ and $O(nh^{d+1})$, and the index structure allows to search for patterns w of length $|w| \leq h$ in optimal time $O(|w| + \mathrm{occ})$.

For larger patterns we need an auxiliary structure, which is a generalized suffix tree (see Section 2.3.1) for the complete input, i.e., all strings in the base set S. The generalized suffix tree $\mathcal{G}(S)$ is linear in the input size and can be built in linear time. We keep the suffix links that are used in the construction process. For a pattern w, we call a substring v right-maximal if $v = w_{i,j}$ is a substring of some string in S, but $w_{i,j+1}$ is not a substring of some string in S. The generalized suffix tree allows us to find all right-maximal substrings of w in time $O(|w|)$. This can be done in the same way as the computation of matching statistics in [CL94]. The approach reminds of the construction of suffix trees: First we compute a canonical reference pair (see Section 3.1.3) for the largest prefix of w that can be matched in the generalized suffix tree. The prefix is right-maximal. Then we take a suffix link from the base of the reference pair replacing the base by the new node. After canonizing the reference pair again, we continue to match characters of w until we find the right-maximal substring starting at the second position in w. This process is continued until the end of w is reached. The total computation takes time $O(|w|)$ because we essentially move two pointers from left to right through w, one for the border of the right-maximal substring and one for the base node of the reference pair. If we build the generalized suffix tree by the algorithm of Ukkonen [Ukk95], this process can also be seen as continuing the algorithm by appending w to the underlying string and storing the lengths of the relevant suffixes.

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

Because we can handle patterns of length at most h efficiently with our main indexing data structure, we only need to search for patterns of length $|w| > h$ using the generalized suffix tree. For each right-maximal substring v of w, we search at all positions where v occurs in any string in S in a prefix of length at most $|w|$, which we can easily find in the generalized suffix tree using bounded value range queries (see Sections 2.3.1 and 2.3.2). For every occurrence, we need to search at most $|w|$ positions, which each takes time $O(d|w|)$. This yields a good algorithm on average if we set $h = c(d + 1)\log n$, where n is the cardinality of S, because the probability to find any right-maximal substring of length $c \log n$ is very small.

Lemma 5.20 (Probability of matching substrings).
Let w be a pattern generated independently and identically distributed at random by a stationary ergodic source satisfying the mixing condition. Let $|w| = (d + 1)l$. The probability that there is a substring u of w of length l that occurs in a prefix of length $|w| + d$ of any string in S is bounded by $cn(|w| + d)|w|e^{-r_{max}l}$ for some constant c.

Proof. For a stationary ergodic source satisfying the mixing condition, the following limit exists [Pit85]:

$$r_{max} = \lim_{n\to\infty} -\frac{\max_{t\in\Sigma^n}\{\ln(\Pr\{t\}) \mid \Pr\{t\} > 0\}}{n} \ . \tag{5.16}$$

(Observe that r_{max} is positive)

Suppose $S = \{s(1), \ldots, s(n)\}$. Set $C_{i,j,k} = \max\{r \mid s(i)_{j,j+r-1} = w_{k,k+r-1}\}$. By stationarity of the source, $\Pr\{C_{i,j,k} > l\} = \Pr\{C_{i,j,1} > l\} = \Pr\{s(i)_{j,j+l-1}\}$. Because $\Pr\{s(i)_{j,j+l-1}\} \leq \max_{t\in\Sigma^l}\Pr\{t\}$, we can apply equation (5.16) and find that $\Pr\{C_{i,j,k} > l\} \leq ce^{-r_{max}l}$ for some constant c and growing l. Hence, we get

$$\begin{aligned}\Pr\{|u| > l\} &= \Pr\left\{\bigcup_{1\leq k\leq|w|,0\leq i\leq n,1\leq j\leq|w|+d} C_{i,j,k} > l\right\} \\ &\leq n(|w| + d)|w|\max_{t\in\Sigma^l}\Pr\{t\} \leq cn(|w| + d)|w|e^{-r_{max}l} \ . \end{aligned}\tag{5.17}$$

□

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

$$cd(|w|)^3(|w| + d)e^{-r_{max}\delta|w|}ne^{-r_{max}c'\ln n} = o(1) \ , \tag{5.18}$$

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

Chapter 6

Conclusion

In this work, we have studied algorithms and data structures for text indexing. We presented the first linear-time algorithm for constructing affix trees. Affix trees have all the capabilities of suffix trees, but they expose more structure of the string for which they are built. They are a compact representation of all repeated substrings of a string, where the tree structure represents the type of repeat, i.e., whether it is maximal, right-maximal, or left-maximal, and the multiplicity it has.

Affix trees are inherently dual. This is reflected in our algorithm, which can construct affix trees on-line, even in both directions. This bidirectional construction may also have applications in studying the local structure of large texts by starting to build an affix tree at a point of interest, expanding to both sides until a desired property is found in the vicinity of the start point.

Finally, affix trees allow a bidirectional search, which is used, e.g., in [MP03]. The paths used in our algorithm may be additionally helpful by hiding "compressible" nodes. The downside of the affix tree data structure is its enormous size (see Section 3.1.4). Whereas a lot of research has already been conducted to reduce the size of the suffix tree (see, e.g., [Kur99, GKS03]), similar work on affix trees remains to be done. For the suffix tree, alternative data structures have also been developed. The suffix array is one such data structure, but an analogous translation of affix trees to affix arrays is not obvious. On the one hand, a node in a suffix tree corresponds to an interval in the suffix array, which is harder to encode. On the other hand, we are working with suffix numbers in suffix arrays; therefore, it is easily possible to derive numbers of shorter or longer suffixes by simple addition or subtraction. Still, we know of no bidirectional search algorithm on suffix arrays that allows searching a pattern by expanding it in both directions.

Our analysis of the average-case running time of the trie-based approach to dictionary indexing is valid for a wide range of string distances. In comparison with the work of Cole et al. [CGL04], our analysis shows that the simple algorithm is already very competitive on average, having a similar asymptotic query time. The analysis also sheds some light on the efficiency of different error models to constrain a search, e.g., Hamming distance versus Hamming distance with don't care characters. We discussed the meaning of our analysis for some applications in Section 4.4. As also mentioned there, one usually considers multiple criteria to guide the search in practice. Although our model captures the influence of don't care characters on Hamming distance, it

cannot satisfactorily answer the question of combining multiple criteria. It would be interesting to see how these interact, e.g., if the search is stopped when any one of a set of multiple distances between a search pattern and the words in the dictionary has become too large.

Besides the issue of determining more parameters such as the variance or convergence properties, another interesting question is the extent to which the results are transferable to more general probabilistic models or to a search in the suffix tree. The latter is of general interest, because a lot of algorithms have been analyzed for the trie model. Some progress has been made on connecting tries to suffix trees [AS92, JS94, JMS04], but the general problem remains open.

Frequently appearing terms in the average-case asymptotic behavior of algorithms on strings are infinite sums involving coefficients of the Gamma function on imaginary points. These lead to small oscillations of bounded amplitude, although it is difficult to give a tight bound. An explanation for this oscillations might follow from the discrepancy between real and integer lengths: The expected length of a common prefix of n independently and uniformly distributed strings over the alphabet Σ is $\log_{|\Sigma|} n$ [Pit85], a real value. On the other hand, strings can only have integer lengths. The amplitude of both oscillations, of $\log_{|\Sigma|} n - \lfloor \log_{|\Sigma|} n \rfloor$ and of the sum in equation (4.62), is exponentially increasing. It would be interesting to derive the deeper relation, thereby simplifying many asymptotics.

For matching with a constant number of errors or mismatches, we described and analyzed a text indexing data structure. It is the first data structure for approximate text indexing that solves this problem with an optimal worst-case query time that is linear in the size of the pattern and the number of outputs. Our approach is very flexible and can be adapted to a number of different text indexing problems. This flexibility is due to its simple tree structure that allows to use range queries for the selection of outputs, a tool that has already been helpful for exact text indexing [Mut02]. Furthermore, we were able to analyze the asymptotic running time for a probabilistic model that is general enough to even include stationary and ergodic Markov chains.

For Hamming distance with a constant number of mismatches d, the asymptotic running time of the trie search algorithm is $O(\log^{d+1} n)$ on average. Because we assumed a long (actually infinite) pattern, this can also be interpreted as $O(m \log^d n)$ for large patterns of length m. On the other hand, the text indexing data structure with (worst-case) query time has (expected) size $O(n \log^d n)$. If one is willing to accept only average-case bounds on the running time, this allows a flexible choice between spending an additional term $O(\log^d n)$ in the running time or in the index size. The best algorithm giving worst-case bounds requires this additional term in the query time and in the index size [CGL04].

For small patterns that have at most logarithmic length $m = O(\log n)$, our algorithm is very efficient in the worst-case and—in contrast to other work—does not require all query patterns to have the same length. For large patterns of size $m = \Omega(\log^d n)$, the method of Cole et al. [CGL04] has an optimal linear query time for constant d. Both data structures require space $O(n \log^d n)$. It remains to be examined whether we can find a data structure with this size and optimal query time for patterns of length $\omega(\log n) \cap o(\log^d n)$. The combination of our methods with the centroid path method of Cole et al. [CGL04] may also yield some new results.

The possibility to "move" the term $O(\log^d n)$ from the complexity of the query

time to the index size also hints towards the existence of a more general time-space trade-off. It would, for instance, be interesting to see if we could design a data structure that is adaptable in the sense that it has size $O(n \log^{d_1} n)$ and supports queries in time $O(m \log^{d_2} n)$ for $d_1 + d_2 = d$ errors.

Furthermore, it would be very interesting to prove a lower bound on the approximate text indexing problem. For the exact case, a linear lower bound has been proven on the index size [DLO01]. However, lower bounds for the approximate indexing problem do not seem easy to achieve. The information theoretic method of [DLO01] seems to fall short because approximate look-up does not improve compression and there is no restriction on the look-up time. Using asymmetric communication complexity, some bounds for nearest neighbor search in the Hamming cube have been shown [BOR99, BR00], but these do not apply to the case where a linear number (in the size of the pattern) of probes to the index is allowed. The lower bound in [BV02] is derived from ordered binary decision diagrams (OBDDs) and assumes that a pattern is always read in one direction.

Finally, the question of practical relevance of the indexing schemes needs further research. Although we believe that our approach is fairly easy to implement, we expect the constant factors to be rather large. We expect even larger constant for the approach used in [CGL04]. Therefore, methods for reducing the space requirements are needed. To this end, it seems interesting to study whether the error sets can be thinned out by including fewer strings. For example, it is not necessary to include strings in the i-th error set that have errors occurring on leaf edges of the $(i-1)$-th error tree. Furthermore, an efficient implementation without trees, based solely on suffix-array-like structures, seems to be possible: Arrays with all leaves are needed for the range minimum queries anyway, and efficient array packing and searching is possible for suffix arrays [AOK02]. For practical purposes, we believe that even a solution for only $d \leq 3$ errors is desirable.

Bibliography

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

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

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

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proc. 2nd Workshop on Algorithms in Bioinformatics (WABI)*, volume 2452 of *Lecture Notes in Computer Science*, pages 449–463. Springer, 2002.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2:53–86, 2004.

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

[AS65] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions, with Formulas, Graphs and Mathematical Tables*. Dover, New York, 1965.

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

[BBH+87] Anselm Blumer, Janet A. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, July 1987.

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 88–94, May 2000.

[BFCP+01] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Least common ancestors in trees and directed acyclic graphs. Preliminary version submitted to the Journal of Algorithms, 2001.

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

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

[BK03] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003.

[BKML+04] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank: update. *Nucleic Acids Research,*, 32(Database issue):D23–D26, 2004.

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

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

[Bra86] Richard C. Bradley. Basic properties of strong mixing conditions. In Ernst Eberlein and Murad S. Taqqu, editors, *Dependence in Probability and Statistics*. Birkhäuser, 1986.

[Bri59] Rene De La Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, pages 295–298, March 1959.

[BT03] Arno Buchner and Hanjo Täubig. A fast method for motif detection and searching in a protein structure database. Technical Report TUM-I0314, Fakultät für Informatik, TU München, September 2003.

[BTG03] Arno Buchner, Hanjo Täubig, and Jan Griebsch. A fast method for motif detection and searching in a protein structure database. In *Proc. German Conference on Bioinformatics (GCB)*, volume 2, pages 186–188, October 2003.

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

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

[BYG90] Ricardo A. Baeza-Yates and Gaston H. Gonnet. All-against-all sequence matching. Technical report, Universidad de Chile, 1990.

[BYG96] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996.

[BYG99] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proc. 6th Int. Symp. on String Processing and Information Retrieval (SPIRE)*, pages 16–23. IEEE, 1999.

[CCGL99] Amit Chakrabarti, Bernard Chazelle, Benjamin Gum, and Alexey Lvov. A lower bound on the complexity of approximate nearest-neighbor searching on the Hamming cube. In *Proc. 31st ACM Symp. on Theory of Computing (STOC)*, pages 305–311, 1999.

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

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

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.

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

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

[Com74] Luis Comtet. *Advanced Combinatorics*. D. Reidel Publishing Company, Dortrecht-Holland, 1974.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.

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

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

[FBY92] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval – Data Structures & Algorithms*, volume 1. Prentice Hall, 1992.

[FGD95] Philippe Flajolet, Xavier Gourdon, and Philippe Dumas. Mellin transforms and asymptotics: Harmonic sums. *Theoretical Computer Science*, 144(1–2):3–58, 1995.

[FGK+94] Philippe Flajolet, Peter J. Grabner, Peter Kirschenhofer, Helmut Prodinger, and Robert F. Tichy. Mellin transforms and asymptotics: Digital sums. *Theoretical Computer Science*, 123:291–314, 1994.

[Fie70] Jerry L. Fields. The uniform asymptotic expansion of a ratio of Gamma functions. In *Proc. Int. Conf. on Constructive Function Theory*, pages 171–176, Varna, May 1970.

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

[FMdB99] Paolo Ferragina, S. Muthukrishnan, and Mark de Berg. Multi-method dispatching: a geometric approach with applications to string matching problems. In *Proc. 31st ACM Symp. on Theory of Computing (STOC)*, pages 483–491. ACM Press, 1999.

[FP86] Philippe Flajolet and Claude Puech. Partial match retrieval of multidimensional data. *J. ACM*, 33(2):371–407, 1986.

[FR85] Philippe Flajolet and Mireille Regnier. Some uses of the Mellin transform in the analysis of algorithms. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI*, pages 241–245. Springer, 1985.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[FS95] Philippe Flajolet and Robert Sedgewick. Mellin transforms and asymptotics: Finite differences and Rice's integral. *Theoretical Computer Science*, 144(1-2):101–124, 1995.

[FS96] Philippe Flajolet and Robert Sedgewick. The average case analysis of algorithms: Mellin transform asymptotics. Technical Report 2956, Institut National de Recherche en Informatique et Automatique (INRIA), 1996.

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

[GBY91] Gaston H. Gonnet and Ricardo A. Baeza-Yates. *Handbook of Algorithms and Data Structures: In Pascal and C.* Addison-Wesley Publishing Company, 2nd edition, 1991.

[GK97] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 2nd edition, 1994.

[GKS03] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software – Practice and Experience*, 33:1035–1049, 2003.

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

[GS04] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Computer and System Sciences*, 69:525–546, 2004.

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

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

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

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

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

[JMS04] Philippe Jacquet, Bonita McVey, and Wojciech Szpankowski. Compact suffix trees resemble PATRICIA tries: Limiting distribution of depth. *J. the Iranian Statistical Society*, 2004.

[JS91] Philippe Jacquet and Wojciech Szpankowski. Analysis of digital tries with Markovian dependency. *IEEE Transactions on Information Theory*, 37(5):1470–1475, 1991.

[JS94] Philippe Jacquet and Wojciech Szpankowski. Autocorrelation on words and its applications: Analysis of suffix trees by string-ruler approach. *J. Combinatorial Theory, Series A*, 66:237–269, 1994.

[JU91] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer, 1991.

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

[Kal02] Olav Kallenberg. Stationary processes and ergodic theory. In *Foundations of Modern Probability*, chapter 9. Springer, 2002.

[Kär95] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. 6th Symp. on Combinatorial Pattern Matching (CPM)*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204, 1995.

[Kär02] Juha Kärkkäinen. Computing the threshold for q-gram filters. In *Proc. 8th Scandinavian Workshop on Algorithm Theory*, pages 348–357, 2002.

[Kin73] John Frank Charles Kingman. Subadditive ergodic theory. *Annals of Probability*, 1(6):883–909, 1973.

[Kir96] Peter Kirschenhofer. A note on alternating sums. *Electronic Journal of Combinatorics*, 3(2), 1996.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming – Fundamental Algorithms*, volume 1. Addison Wesley, 3rd edition, September 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming – Seminumerical Algorithms*, volume 2. Addison Wesley, 3rd edition, July 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, February 1998.

[KP86] Peter Kirschenhofer and Helmut Prodinger. Some further results on digital search trees. In Laurent Kott, editor, *13th Int. Colloq. on Automate, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, Rennes, France, 1986. Springer.

[KPS93] Peter Kirschenhofer, Helmut Prodinger, and Wojciech Szpankowski. Multidimensional digital searching and some new parameters in tries. *Int. J. Foundations of Computer Science*, 4:69–84, 1993.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming (ICALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.

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

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

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

[ŁS97] Tomasz Łuczak and Wojciech Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Transactions on Information Theory*, 43(5):1439–1451, 1997.

[LS99] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Dept. of Comp. Science, Lund University, Sweden, 1999.

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

[Maa00] Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. In *Proc. 11th Symp. on Combinatorial Pattern Matching (CPM)*, volume 1848 of *Lecture Notes in Computer Science*, pages 320–334. Springer, June 2000.

[Maa03] Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, June 2003.

[Maa04a] Moritz G. Maaß. Average-case analysis of approximate trie search. In *Proc. 15th Symp. on Combinatorial Pattern Matching (CPM)*, volume 3109 of *Lecture Notes in Computer Science*, pages 472–484. Springer, July 2004.

[Maa04b] Moritz G. Maaß. Average-case analysis of approximate trie search. Technical Report TUM-I0405, Fakultät für Informatik, TU München, March 2004.

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

[MF04] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, June 2004.

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

[MN04] Moritz G. Maaß and Johannes Nowak. A new method for approximate indexing and dictionary lookup with one error. *To be published in ipl Information Processing Letters (IPL)*, 2004.

[MN05a] Moritz G. Maaß and Johannes Nowak. Text indexing with errors. Technical Report TUM-I0503, Fakultät für Informatik, TU München, March 2005.

[MN05b] Moritz G. Maaß and Johannes Nowak. Text indexing with erros. In *Proc. 16th Symp. on Combinatorial Pattern Matching (CPM)*, To be published in Springer LNCS, 2005.

[Mor68] Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, 1969.

[MP03] Giancarlo Mauri and Giulio Pavesi. Pattern discovery in RNA secondary structure using affix trees. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 278–294, 2003.

[MPP01] Conrado Martinez, Alois Panholzer, and Helmut Prodinger. Partial match queries in relaxed multidimensional search trees. *Algorithmica*, 29:181–204, 2001.

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

[MZS02] Krisztián Monostori, Arkady Zaslavsky, and Heinz Schmidt. Suffix vector: Space- and time-efficient alternative to suffix trees. In Michael J. Oudshoorn, editor, *Proc. Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002.

[Nav98] Gonzalo Navarro. *Approximate Text Searching.* PhD thesis, Dept. of Computer Science, University of Chile, Santiago, Chile, 1998.

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

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

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

[Nör24] Niels Erik Nörlund. *Vorlesungen über Differenzenrechnung*. Springer, Berlin, 1924.

[Now04] Johannes Nowak. A new indexing method for approximate pattern matching with one mismatch. Master's thesis, Fakultät für Informatik, Technische Universität München, Garching b. München, Germany, February 2004.

[Ofl96] Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, 1996.

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

[Pit86] Boris Pittel. Paths in a random digital tree: Limiting distributions. *Advances in Applied Probability*, 18:139–155, 1986.

[PW99] Edward H. Porter and William E. Winkler. Approximate string comparison and its effect on an advanced record linkage system. In *Record Linkage Techniques—1997: Proceedings of an Int. Workshop and Exposition*, pages 190–199, Washington, D.C., 1999. National Research Council, National Academy Press.

[Rud87] Walter Rudin. *Real and Complex Analysis*. McGraw-Hill, 3rd edition, 1987.

[SM02] Klaus U. Schulz and Stoyan Mihov. Fast string correction with Levenshtein automata. *Int. J. on Document Analysis and Recognition (IJDAR)*, 5:67–85, 2002.

[Sto95] Jens Stoye. Affixbäume. Diplomarbeit, Universität Bielefeld, May 1995.

[Sto00] Jens Stoye. Affix trees. Technical Report 2000-04, Universität Bielefeld, Technische Fakultät, 2000.

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

[Szp88a] Wojciech Szpankowski. The evaluation of an alternative sum with applications to the analysis of some data structures. *Information Processing Letters (IPL)*, 28:13–19, 1988.

[Szp88b] Wojciech Szpankowski. Some results on v-ary asymmetric tries. *J. Algorithms*, 9:224–244, 1988.

[Szp93a] Wojciech Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, September 1993.

[Szp93b] Wojciech Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comp.*, 22(6):1176–1198, December 1993.

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

[TE51] Francesco Giacomo Tricomi and Arthur Erdélyi. The asymptotic expansion of a ratio of Gamma functions. *Pacific J. Mathematics*, 1:133–142, 1951.

[Tem96] Nico M. Temme. *An Introduction to Classical Functions of Mathematical Physics.* Wiley, New York, 1996.

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

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

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

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[WDH+04] F.A.O. Werner, G. Durstewitz, F.A. Habermann, G. Thaller, W. Krämer, S. Kollers, J. Buitkamp, M. Georges, G. Brem, J. Mosner, and R. Fries. Detection and characterization of SNPs useful for identity control and parentage testing in major European dairy breeds. *Animal Genetics*, 35(1):44–49, February 2004.

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

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

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

Index

"accelerator", **5**
active prefix, **36**
active suffix, **36**
affix tree, 2–4, 8, **34**, 31–62, 115
  compact, 3, 36
  example of, 32
amortized, 16–17
  analysis, **16**, 31, 46, 54, 56, 57
  constant time, **16**
average compactification number, **77**, 79
average-case analysis, 6–9, 15, 63–65, 67

base set, **95**, 108, 113
Bernoulli numbers, **28**, 81
  generalized, **28**, 82
Bernoulli polynomials, generalized, **28**, 78
Bernoulli trials, independent, 17, 65
  see also memoryless source
Beta function, 27, 69, 70
  approximation of, 74
bidirectional, 54
  construction of affix trees, 54–62
  search, 115
Borel-Cantelli Lemma, 66, 67

canonize, **42**, 41–43, 113
Cartesian tree, **23**
Cauchy Residue Theorem, 27
centroid path, 8, 94, 116
complexity, **15, 16**
  expected, 18
  measures, 15–16
convergence, 18–19
  almost sure, **19**, 66, 67
    method of Kesten and Kingman, 66
  in probability, 6, **18**, 66
cost model, 15–16
  bit, **16**
  logarithmic, **15**
  uniform, **15**
  word, **16**
  see also complexity
CST, **20**

DAWG, 3, 7
  see also directed acyclic word graph
decanonize, **46**, 43–47
depth, **13**
  $\mathrm{depth}()$, **13**, 100
  see also string depth
depth first traversal, 6, 25, 93, 109, 110
dictionary indexing, 4, 5
  approximate, 5, 8, 9, 26, 63, 93, 115
  problem, **25**
directed acyclic word graph, 22
  compact, 3
  see also DAWG
distance, 14–15
  arithmetic, **90**, 91
  edit, 2, 6–9, **15**, 26, 94–97, 99, 101, 108
    k-bounded computation of, **99**, 102, 108
    weighted, **14**
  Hamming, 2, 5, 7, 8, **14**, 26, 64, 88, 90, 91, 95, 96, 115, 116
  Levenshtein, 15
  see also string distance
document listing problem, **22**, 25, 26
don't-care symbol, 6, **14**, 90
dynamic programming, 14, 24
dynamic table, **16**, 35, 39, 49

edge, 13
  atomic, **13**, 38
  compact, 20
  see also open edge
edit distance, *see* distance
edit graph, **96**, 97, 99
  relevant, 96, **97**, 97–99, 102, 114
edit operation, **14**
edit path, **97**
edit sequence, *see* ordered edit sequence
edit stage, **97**
end node, **49**
end point, **42**, 50
ergodic, **18**
error tree, **104**
  height of, 94, 102, 111
Euler Tour, 24
Eulerian numbers, 29
Eulerian polynomials, **28**, 82
explicit location, **13**, 32, 33
exponential cancellation effect, **26**, 76

Gamma function, 27, 70

Hamming distance, *see* distance
height, **13**

implicit location, **13**, 34, 35, 43

k-d-trie, 6
k-error occurrence, **101**

Law of Large Numbers, 65
left-branching, **12**
Levenshtein distance, *see* edit distance
linear search algorithm, 64
longest common subsequence, **14**

Markov source, **17**
matching statistics, 113
Mellin transform, 6, 28, 29, 84
memoryless source, **17**, 64, 92
mixing condition, **18**, 19, 111, 114

nested, **12**
  longest $\sim$ prefix, *see* active prefix
  longest $\sim$ suffix, *see* active suffix
node depth, 13, 24
non-nested, *see* nested
"normal iteration", *see* suffix iteration

occurrence, **11**
  approximate, **25**
  set of, **12**
  $\mathrm{occurrences}()$, **12**
  see also k-error occurrence
occurrence listing problem, **25**
open edge, **36**, 39, 41, 44, 49, 51
ordered edit sequence, **97**, 97–99
output sensitive, **22**, 93

path, **13**, **38**
  of a Σ^+-tree node, **13**
  $\mathrm{path}()$, **13**, 100
  of prefix only nodes, 4, **38**, 39, 47, 49–51, 54, 56, 115
path compression, **13**, 20
PATRICIA tree, **20**
  example of, 21
pattern, **11**
prefix, **11**
  active, *see* active prefix
  set of, **12**
  $\mathrm{prefixes}()$, **12**
prefix free, **12**
  reduction, **12**
  set, **12**, 20, 21, 101
  $\mathrm{pfree}()$, **12**
prefix iteration, **47**
prefix node, **35**
prefix only node, **38**
prefix tree, 3, **34**
purse, 57

range minimum query, **23**
range query, 9, 22–25, 101, 102, 109, 110, 116, 117
  bounded value, **23**, 24, 109, 114
  colored, **23**, 24, 109
  geometric, 7
reference pair, **35**, 113
  base of, **35**
  canonical, **35**, 36, 113
  combined, 35, 36
  prefix, 35, 36
  representation of virtual nodes, 13

size of, 39
suffix, 35, 36
“register”, **5**
relevant suffix, **41**, 113
residue, **27**
“reverse iteration”, *see* prefix iteration
Rice’s integrals, **26**, 26–29, 69, 76, 78, 84
right-branching, **12**

sentinel, **20**, 22, 103, 109
σ-field, 17, 18
σ, **64**
Σ, **11**
Σ^+-tree, **13**, 19–22, 32–34, 95, 99, 100
atomic, **13**
see also trie
compact, **13**
see also PATRICIA tree
weak, **100**
see also weak trie
stationary, **18**
stationary and ergodic, **17**
Markov chain, 18, 19, 116
sequence, 18, 19
source, **17**, 18, 19, 110, 111, 114
stem from, **103**, 105, 111
Stirling’s formula, 70, 71, **75**
string, **11**, 11–12
depth, **13**
distance, 14–15, 115
comparison-based, **14**, 63, 64, 90, 91
length, **11**
prefix of, **11**
suffix of, **11**
strong α-mixing condition, **18**, 19
substring, **11**
proper, **12**
set of, **12**
$\mathrm{psubstrings}()$, **12**
set of, **12**
$\mathrm{substrings}()$, **12**
subtree, **13**
suffix, **11**
active, *see* active suffix
nested, *see* nested
proper, **12**
set of, **12**, 95, 110
$\mathrm{suffixes}()$, **12**
suffix array, 2, 4–6, 22, 115, 117
suffix cactus, 22
suffix iteration, **47**
suffix link, 3, 4, **20**, 31–34, 113
atomic, 4
chain of, 32, 33, 37
non-atomic, 37
suffix link tree, **32**
example of, 32
of atomic suffix tree, 32
of compact suffix tree, 33
suffix node, **34**
suffix tree, 2–9, **20**, 92, 94, 99, 100, 115
atomic, 3, 32, 40
duality of, 32
compact, 3, 20
weak duality of, 32
construction of, 40–47
Ukkonen, 40
Weiner (modified), 43
“dual”, 34
example of, 21, 32
generalized, **22**, 91, 93, 113, 114
for document listing, 22–25
height of, 19, 94, 111
suffix vector, 22

text corpus, **11**, 26, 93
text indexing, 2, 11, 19, 115
approximate, 2, 4–8, **25**, 93–95, 116, 117
problem, 7, 9, **22**, 25–26, 93, 116
classification of, **26**, 63, 64, 93, 95, 108
trie, 5, 6, 8, 9, **20**, 26
compact, **20**, 99–101, 104
“compactification” of strings, 77
compressed, 20
see also PATRICIA tree
example of, 21
height of, 90, 100, 102, 111
“hidden” characters, 77

random, 69
search algorithm, 65
number of visited nodes, 67
see also k-d-trie

underlying string, **35**
unique branching criterion, **13**, 33, 100, 109
relaxed, **100**

virtual node, **13**, 35, 100, 109

weak trie, 9, **100**, 99–100, 104, 105, 109
with high probability, **18**, 19, 93, 110–112
word set of a Σ^+-tree, **13**
$\mathrm{words}()$, **13**, 100