Department of Computer Science
Series of Publications A
Report A-1999-4

# Repetition-Based Text Indexes

Juha Kärkkäinen

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1999-4

# Repetition-Based Text Indexes

Juha Kärkkäinen

*To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium, Teollisuuskatu 23, on December 3rd, 1999, at 12 o'clock noon.*

University of Helsinki
Finland

## Contact information

Postal address:

Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

ISSN 1238-8645
ISBN 951-45-8917-3
Computing Reviews (1998) Classification: F.2.2, E.1
Helsinki 1999
Helsinki University Printing House

## Repetition-Based Text Indexes

Juha Kärkkäinen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Juha.Karkkainen@cs.Helsinki.FI
http://www.cs.Helsinki.FI/Juha.Karkkainen/

PhD Thesis, Series of Publications A, Report A-1999-4
Helsinki, November 1999, 106 pages
ISSN 1238-8645, ISBN 951-45-8917-3

### Abstract

*Repetition-based indexing* is a new scheme for preprocessing a text to support fast pattern matching queries. The scheme provides a general framework for representing information about repetitions, i.e., multiple occurrences of the same string in the text, and for using the information in pattern matching. Well-known text indexes, such as suffix trees, suffix arrays, DAWGs and their variations, which we collectively call *suffix indexes*, can be seen as instances of the scheme.

Based on the scheme, we introduce the *Lempel–Ziv index*, a new text index for string matching. It uses the repetition information in a *Lempel–Ziv parse*, which is a division of the text into non-overlapping substrings with earlier occurrences, and which is also used in the Ziv–Lempel family of text compression methods. The Lempel–Ziv index offers a possibility for a space–time tradeoff. The space requirement can be smaller than for suffix indexes by up to a logarithmic factor, while the query time is larger but still sublinear in the length of the text. The only previous text index offering a space–time tradeoff is the sparse suffix tree. The Lempel–Ziv index improves on the results of the sparse suffix tree in many cases.

Text indexes for $q$-gram matching, i.e., for matching string patterns of length $q$, are used in some approximate string matching algorithms. We introduce a new repetition-based $q$-gram index, the *Lempel–Ziv index for $q$-grams*, that has asymptotically optimal space requirement and query time provided that $q$ is a constant or grows slowly enough with respect to the length of the text. Queries are as fast as with traditional $q$-gram indexes, but the space requirement can be smaller by a logarithmic factor.

Some additional novel data structures are developed for subproblems arising in the Lempel–Ziv indexing methods. These include a new variation of the suffix tree with a faster query time, a variation of a data structure for two-dimensional range searching with new possibilities for space–time tradeoffs, and a new data structure, called the *nesting leveled list*, for the range containment problem.

**Computing Reviews (1998) Categories and Subject Descriptors:**

F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—pattern matching, sorting and searching, geometrical problems and computations

E.1 Data Structures—trees

**General Terms:**

Algorithms, Theory

**Additional Key Words and Phrases:**

String matching, text indexes, Lempel–Ziv parsing, $q$-grams, range searching

# Acknowledgements

I am deeply grateful to my supervisor, Professor Esko Ukkonen, for his guidance, encouragement and support throughout my studies. He introduced me to the research area of string algorithms, and many of the main ideas of the research were created in collaboration with Esko. I have also had the pleasure to work with Dr. Erkki Sutinen during the research. A significant portion of the results are from a joint publication with Erkki.

The research has been carried out at the Department of Computer Science of the University of Helsinki. The department, headed by Professors Martti Tienari, Esko Ukkonen and Timo Alanko at different times during the research, has provided me with excellent working conditions. Many colleagues and friends have contributed to making the department a great place to work at. They have also helped me to keep going by frequently inquiring about the progress of my thesis.

The Academy of Finland and the Helsinki Graduate School of Computer Science and Engineering have provided financial support.

# Contents

# Chapter 1

# Introduction

*Repetition-based indexing* is a new scheme for preprocessing a text to support fast pattern matching queries. It is quite unlike any previous text indexing method. The basic idea of the scheme is to take advantage of information about repetitions in the text. The scheme is very general and can, in principle, be used in any type of pattern matching in strings. Concrete variations are given here only for basic string matching and its special case, $q$-gram matching.

## 1.1 String matching

Let us first look at the problem of *string matching*: given a short string $P$, called the *pattern*, and a long string $T$, called the *text*, the task is to find all occurrences of $P$ in $T$. This fundamental problem has many variations. The variation of interest here is the *indexed* (or offline or static) string matching, where the text is preprocessed, independently of the pattern, to obtain a *text index* that supports fast *pattern queries*. The main parameters of a solution to the indexed string matching problem are the space requirement of the index and the query time. The time and space needed for building the text index are less important. Note that we want to find all occurrences of any pattern; for example, finding only the occurrences as a word of a natural language text would be an easier task.

A related problem is the *prefix matching problem*: given a set $V$ of strings, and a query string $Q$, find the strings in $V$ that have $Q$ as a prefix. The string matching problem can be formulated as a prefix matching problem: $V$ is the set of all suffixes of the text $T$, and $Q$ is the pattern $P$. The idea is that, if a suffix has the pattern $P$ as a prefix, the starting position of the suffix is also the starting position of an occurrence of $P$.

All efficient text indexes for the string matching problem are based on the reformulation as a prefix matching problem. In other words, they are basically data structures that store a set of strings (the suffixes) and support prefix matching queries. Such data structures, which we call *suffix indexes*, include the suffix tree [Wei73], which is a compressed trie (PATRICIA trie [Mor68]) of the suffixes; the suffix array or the PAT array [MM93, GBYS92], which is an ordered array of the suffixes; and the suffix automaton or DAWG (directed acyclic word graph) [BBH$^+$85, Cro86], which is an automaton recognizing the suffixes. There are also many variations of these data structures [BBH$^+$87, AN95, Kär95, Irv95, CD96, CV97, Kur98, GKS99].

Using the suffix indexes, string matching queries can be answered very quickly. For example, the query time of a typical implementation of a suffix tree is $O(|P||\Sigma| + k)$ or $O(|P| \log |\Sigma| + k)$, where $\Sigma$ is the alphabet and $k$ is the number of occurrences of the pattern $P$. In fact, the query time can be reduced to $O(|P| + k)$. This is effectively optimal, since any solution needs to inspect all characters of the pattern and to report all occurrences (however, see Section 1.6 for the effect of bit-packing).

The space requirement of the suffix indexes can be a problem. Reducing the space requirement (by a constant factor) has been a main theme in the recent developments on the suffix indexes (see, e.g., [AN95, Kär95, CD96, CV97, Kur98, GKS99]). All the suffix indexes and their variations need to have $\Omega(|T|)$ pointers to the text to represent the suffixes.[1] In fact, the smallest of the data structures, the suffix array in its simplest form, consists only of an array of $|T|$ pointers and the text itself. This might seem to be asymptotically optimal, because any text index needs to store the $|T|$ characters of the text (or the equivalent information in some form). However, a character of the alphabet $\Sigma$ can be packed into $\lceil \log |\Sigma| \rceil$ bits. Thus the text can be stored using just $O(|T| \log |\Sigma|)$ bits, while the pointers require $\Omega(|T| \log |T|)$ bits. For a large text and a small alphabet, the difference can be significant. For example, a suffix array for the human genome ($|T| \approx 3 \cdot 10^9$, $|\Sigma| = 4$) requires at least 16 times as much storage as the genome itself.

To get below the space requirement of $\Omega(|T| \log |T|)$ bits, we have to look for a different approach to indexing the text. In the extreme case, we can have no index, just the text itself. Then string matching is performed by scanning the text through with a *sequential matching algorithm*, but the resulting linear query time can be too high. The first method with a space–time tradeoff in between the suffix indexes and sequential matching

[1]The compact DAWG [BBH$^+$87, CV97] can survive with a sublinear number of pointers if the text is highly repetitive, but not in the worst case, on average [BEH89] or in practice [CV97].

algorithms is the sparse suffix tree [KU96b]. It is a simple variation of the suffix tree that stores only every $h$th suffix of the text. As $h$ increases, the space requirement decreases, but it becomes exponentially harder to find all occurrences (see Section 3.4.2). With different choices for $h$, different space–time tradeoffs can be achieved.

## 1.2 Repetition-based indexing

The basic idea of repetition-based indexing is to take advantage of information about repetitions in the text; by repetitions we mean multiple occurrences of the same string in the text. If we have found an occurrence of the pattern, we may be able to locate other occurrences using the repetition information only.

The execution of a query consists of two phases. The first phase, *primary search*, searches the text areas where repetition information is not available. The occurrences found in this phase are called *primary occurrences*. Then the second phase, *secondary search*, takes the primary occurrences as input and uses precomputed information about the repetitions to find the other occurrences, called *secondary occurrences*.

This general framework allows a wide variety of different realizations. In fact, suffix indexes and sequential matching algorithms can be seen as extreme instances of this scheme. Sequential matching algorithms use no index and no precomputed information about repetitions; there are no secondary occurrences and no secondary searching phase, just a slow primary searching phase. Suffix indexes, on the other hand, contain information about all repetitions in the text in an accessible form. String matching queries with suffix indexes have a quite natural division into the primary and the secondary searching phases. For example, the primary searching phase of suffix trees finds the node $v$ that represents the pattern. The leaves that are descendants of $v$ represent the suffixes that have the pattern as a prefix. The secondary search traverses the subtree rooted at $v$ to find the leaves.

## 1.3 Lempel–Ziv index

Consider the text divided into a sequence of (nearly) consecutive non-overlapping blocks such that the substring in each block also occurs somewhere else in the text. We call such a division a *Lempel–Ziv (LZ) parse*, because it was first described by Lempel and Ziv [LZ76]. LZ parses are in a key role in the Ziv–Lempel text compression methods [ZL77, ZL78] and their many variations [Sto88, BWC89, BCW90].

A repetition-based index that uses the repetition information in an LZ parse is called a *Lempel–Ziv (LZ) index*. The occurrences of the pattern completely inside a block are the secondary occurrences to be found in the secondary searching phase using the repetition information. The occurrences that are not inside any block, i.e., overlap a *boundary* between the blocks, are the primary occurrences.

We defer the detailed description of LZ indexing until Chapter 5, but primary searching deserves a closer look here. For primary searching, we want to have an index of the text regions surrounding the boundary positions. The middle between the boundaries, i.e., the inside of the blocks, is taken care of by the secondary search.

Let us return to the sparse suffix tree. It provides an efficient index of the text regions that are in the beginning of the stored suffixes, i.e., to the immediate right of the starting positions of the suffixes. The most poorly represented text regions are to the immediate left of the starting positions. This was recognized in [KU96b], and it was suggested to take also the prefixes of the text ending at the same positions and to put their reverses into a dual sparse tree. This is not very useful in sparse suffix trees; it just moves the poorly represented regions from the left of the starting positions to the middle between the starting positions. However, in an LZ index the prefix–suffix pairs are just what is called for to represent the surroundings of the boundary positions.

## 1.4 Range searching

We have seen that the string matching problem can be reduced to the prefix matching problem. The prefix matching problem, in turn, is a special case of the *range search problem*: given a subset $V$ of some universe $U$ and a query range $[l, h]$ of $U$, find the elements of $V$ that are in the range $[l, h]$. In prefix matching, the universe $U$ is the set of strings and the query range is the range of strings with the query string $Q$ as a prefix. Thus string matching is range searching, which is also obvious in suffix arrays.

Now consider the primary search problem in LZ indexing. We want to find the occurrences of the pattern overlapping some boundary position. For such boundaries, a pattern prefix is a suffix of the text prefix ending at the boundary, and a pattern suffix is a prefix of the text suffix starting at the boundary. Both the text prefixes matching a given pattern prefix and the text suffixes matching the corresponding pattern suffix are answers to range queries. We can interpret the prefix–suffix pair representing a boundary position as a pair of coordinates in a two-dimensional universe.

Then the primary search problem is, in fact, a two-dimensional (orthogonal) range search problem. Thus two-dimensional range searching is the key subproblem in LZ indexing. In fact, it is the problem that dominates the time and space requirements in most cases.

## 1.5 Lempel–Ziv index for $q$-grams

The *$q$-gram matching problem* is a special case of string matching, where the pattern $P$ is restricted to be a *$q$-gram*, a string of length $q$. A text index for $q$-gram matching is called a *$q$-gram index*. Some approximate string matching algorithms use a $q$-gram index in a filtering phase to eliminate the text areas that cannot contain an occurrence [JU91, HS94, Mye94, ST96, NBY98].

In a usual implementation of the $q$-gram index, the $q$-grams occurring in the text are stored in a data structure such as a trie, a hash table or a look-up table. Associated with each $q$-gram is a list of its occurrences in the text. Queries can be answered in $O(q + k)$ time (or even $O(1 + k)$ time if the query $q$-gram is provided in a suitable form, e.g., coded as an integer), where $k$ is the number of occurrences. The data structure for storing the $q$-grams can be quite small if $q$ is small. However, the lists of occurrences require $\Omega(|T| \log |T|)$ bits.

Once again, the above $q$-gram index fits the repetition-based indexing scheme with natural primary (find the start of the occurrence list) and secondary (scan the list) searching phases. In a new repetition-based indexing method, the *Lempel–Ziv (LZ) index for $q$-grams*, the primary search remains the same, but the occurrence lists are replaced with a data structure based on a variation of the LZ parse.

In the variation of the LZ parse, consecutive blocks overlap by $q - 1$ positions. Thus it contains all information about $q$-gram repetitions, i.e., all information in the occurrence lists. However, if $q$ is small enough, the LZ parse can be stored in $\Theta(|T| \log |\Sigma|)$ bits while still being able to find all occurrences in $O(1 + k)$ time.

## 1.6 Results and contributions

The main contributions of this dissertation are the repetition-based indexing scheme and its two realizations, the LZ indexes for string matching and for $q$-gram matching. The repetition-based indexing scheme provides a new way of looking at pattern matching problems, and might have useful applications beyond what is described here.

The model of computation in this dissertation is the *word RAM* [Hag98], a random access machine with unit-cost operations and a word size $\Theta(\log |T|)$. The bound on the word size is necessary for *bit-parallel* or *word-based* algorithms, which take advantage of the implicit bit-level parallelism in operations on words, and for *bit-packing*, which means using every bit of a word efficiently to store data. Due to systematic use of bit-packing, the space requirements are measured in bits.

Bit-packing is significant for string algorithms, because it not only reduces the space requirement of strings, but also, as an application of bit-parallelism, enables $\Theta(\log |T| / \log |\Sigma|)$ character comparisons in one step. This leads to a new variation of the suffix tree that supports queries in time $O(|P| \log |\Sigma| / \log |T| + \log \log |T| + k)$. Note, however, that the query time of $O(|P| + k)$ may still be better for short patterns. Both of the mentioned variations have the space requirement of $O(|T| \log |T|)$ bits.

The space requirement and the query time of the LZ index depends on the choice of the two-dimensional range searching method; a faster query time comes at the cost of a larger space requirement. As an example, the query time

$$O\left(\frac{|P|^2 \log |\Sigma|}{\log |T|} + |P||T|^{\varepsilon / \log |\Sigma|} + k\right),$$

where $\varepsilon$ is an arbitrary positive constant, can be achieved with space requirement of $O(|T| \log |\Sigma|)$ bits, the same as needed for the text itself. The sparse suffix tree with the same space requirement has the query time $O(|P||T|^{\varepsilon} + k)$, which can be much larger for long texts.

The LZ index for $q$-gram matching is asymptotically optimal both in space ($O(|T| \log |\Sigma|)$ bits, the space requirement of the text itself) and in time ($O(1 + k)$), provided that $q$ is a constant or grows slowly enough with respect to the length of the text. The space requirement is smaller than with traditional $q$-gram indexes by the factor $\Omega(\log_{|\Sigma|} |T|)$. Also, if the text is compressible, the space requirement can be still smaller. This is not true for the LZ index for string matching, because it needs the uncompressed text.

The LZ indexing methods involve a number of basic computational geometry problems as subproblems, for which we present some new results. For two-dimensional range searching, we give a variation of a data structure by Overmars [Ove88] that offers new possibilities for space–time tradeoffs. A new data structure called the *nesting leveled list* is in central role in the LZ index for $q$-grams. It solves the range containment problem, finding the ranges that contain a given query range.

Many of the ideas and results of this dissertation were developed together with Esko Ukkonen and Erkki Sutinen, and have been published in [KU96a] and [KS98]. The LZ index for string matching was developed in [KU96a]. Many of the details have changed since the original publication, though. The LZ index for $q$-grams was developed in [KS98] in a more or less the same form as described here. The main ideas of repetition-based indexing were present in those works, although the general framework was not described.

## 1.7 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we describe the computational model, define basic concepts, and introduce the pattern matching problems and a number of subproblems arising in repetition-based indexing. Solutions to the basic problems, mostly previously published but some new variations, too, are given in Chapter 3.

The general repetition-based indexing scheme is presented in Chapter 4. Some examples of the scheme, in particular a suffix index, are given. The LZ index for string matching is described in Chapter 5 and for $q$-gram matching in Chapter 6. Some concluding remarks are given in Chapter 7.

# Chapter 2

# Preliminaries

In this chapter, we lay the formal foundation for the dissertation. First, we describe the computational model, discuss some of its details, and explain the reasons for choosing it. Then we introduce the main concepts and the related notation and terminology. Finally, we define the main problems and subproblems encountered later, and discuss their properties and relations.

## 2.1 Model of computation

The model of computation used in this dissertation is the *word RAM* [Hag98], a random access machine with unit-cost operations and logarithmic word size. In other words, the word RAM has $2^{O(w)}$ registers each of which stores a $w$-bit word. All operations on words that are usually available on modern computers (memory access, flow control, comparisons, basic arithmetic operations, bitwise shift and Boolean operations, etc.) can be performed in constant time.

The word RAM differs from the standard RAM mainly in its bound on the word size. This is not a severe restriction in the sense that most traditional algorithms work on the word RAM with the same time complexity as on the traditional RAM. However, the so called *bit-parallel* or *word-based* algorithms, which take advantage of the implicit bit-level parallelism in operations on words, require a bound on the word size, because otherwise unrealistic speedups would be possible. There has been a lot of recent activity on bit-parallelism. On the theoretical front, the upper bounds on many of the most basic searching and sorting problems have been improved; see [Hag98] for a survey. On the practical side, bit-parallel methods have proved to be the fastest in practice in many string matching problems; see, e.g., [Nav98, Mye98, BYN99] for recent results on approximate string matching.

Bounded word size is also necessary with *bit-packing*, which means using every bit of a word efficiently to store data. For example, several small objects can be packed into one word. With constant time bitwise shift and Boolean instructions available, bit-packing causes only constant factor overhead on operations. Sometimes bit-packing may even reduce the time requirement. For example, bit-packed strings of length $m$ over an alphabet of size $c$ can be compared in $O(m \log c/w)$ time. In this dissertation, bit-packing plays a central role as shown by the following example.

**Example 2.1** By traditional conventions, a string of length $m$ over an alphabet of size $c$ and its suffix tree both require $O(m)$ words. However, the string can be packed in $O(m \log c/w)$ words, while the suffix tree needs $\Omega(m \log m/w)$ words to store $\Omega(m)$ pointers to the string. Thus, the suffix tree may require asymptotically more space than the text itself unlike is commonly assumed.

The word size $w$ is often an independent parameter unrelated to the space requirement; see [Hag98] for a discussion. We, however, choose the word size $w$ to be $\Theta(\log s)$, where $s$ is the maximum space requirement of the task. The lower bound $w = \Omega(\log s)$ is due to the need to fit a pointer into a constant number of words. The upper bound is a useful convention that often avoids explicit inclusion of $w$ in time and space requirements. For tasks with large space requirement, such as indexed string matching, the assumption $w = \Theta(\log s)$ is quite realistic. Note, though, that $s$ is the space requirement of the whole task; a subtask may require much less space. Therefore, in many of the subproblems in this dissertation, we will not assume any upper bound on the word size.

When using bit-packing and not knowing an upper bound for $w$, $w$ must be included explicitly in space requirements. To avoid this, we give space requirements in bits. This also avoids some counterintuitive space requirements, as shown by the following example.

**Example 2.2** The approximate string matching algorithm of Chang and Lawler [CL94] builds a suffix tree for a pattern of length $m$ and uses it to find the approximate occurrences of the pattern in a text of length $n$, where usually $n \gg m$. Let us assume that this is the "whole task", i.e., that $w = \Theta(\log n)$. For simplicity, let the alphabet size be constant. Then, the suffix tree fits into $\Theta(m \log m/ \log n)$ words. Now the space requirement of the suffix tree depends on the length of the text although the suffix tree itself is independent of the text. This is avoided if the space requirement of the suffix tree is given as $O(m \log m)$ bits.

To avoid confusion, we will always mention the unit (usually bits) with a space requirement.

Let us discuss one more aspect of the computational model, the instruction set. We assume that all instructions usually available on modern computers can be performed in constant time. Of the usual instructions, multiplication and especially division are commonly excluded, because they are not in the circuit complexity class $AC^0$ of instructions implementable with a constant depth, unbounded fan-in Boolean circuits of size $w^{O(1)}$ [KR90]. Division is considered to be more expensive than multiplication. There are also results in the literature that use nonstandard $AC^0$ instructions, i.e., instructions that are in $AC^0$ but not commonly available in real computers [AMRT96, Hag98, AMT99]. We do not use such instructions.

In this dissertation, division is not needed for any of the results. Multiplication, on the other hand, is needed in a key method, perfect hashing (Lemma 3.3), and in the many results dependent on it. Perfect hashing or an equivalent method cannot be implemented using just $AC^0$ instructions [AMRT96]. A common operation in this dissertation, random access to a bit-packed array of $b$-bit objects, needs multiplication by $b$ and division by $w$. However, if necessary, we can assume that $b$ and $w$ are powers of two; then the multiplication and division can be implemented as bitwise shifts.

There is one more convention used in this dissertation that differs from common practice. We explicitly show the effect of parameters such as the alphabet size that are often excluded because they are assumed to be constant. For example, a string of length $m$ over an alphabet of size $c$ has a space requirement of $O(m \log c)$ bits. There are two reasons for this convention. First, the assumption that such parameters are constant might not always hold. Without such assumptions, the results are more general. Second, there is a big difference between alphabet sizes 2 and $2^{32}$ although both are constants. Such differences become visible when the parameters are shown explicitly. This is illustrated in the following example.

**Example 2.3** Consider a suffix tree for a text of length $n$ over an alphabet of size $c$ and a query with a pattern of length $m$. The size of the suffix tree and the query time depend on the implementation of the branching (accessing the children of a node) in the suffix tree. If the branching is implemented with linked lists, the size is $O(n \log n)$ bits and the query time is $O(cm)$. If the branching is implemented with balanced trees, the size is $O(n \log n)$ bits and the query time is $O(m \log c)$. If the branching is implemented with table lookup, the size is $O(cn \log n)$ bits and the query time is $O(m)$. If $c$ is assumed to be constant and is excluded from the time and space requirements, there is no difference between the three implementations. However, for large alphabets the differences are significant in practice.

## 2.2 Basic definitions

Next we define basic concepts, terminology and notation related to strings and ranges.

### 2.2.1 Strings

**Definition 2.4** An *alphabet* $\Sigma$ is a finite ordered set. The elements of $\Sigma$ are called *characters*. For $n > 0$, the Cartesian product of $n$ $\Sigma$'s, denoted by $\Sigma^n$, is the set of *strings* of length $n$ over $\Sigma$. The set $\Sigma^0$ contains one element, the empty string $\epsilon$. The set of all strings over $\Sigma$ is $\Sigma^* = \cup_{n \geq 0} \Sigma^n$.

In other words, a string is a finite sequence of characters. We use the convention that strings are denoted by capital letters and the characters of a string by the corresponding lower case letter with a running subscript starting from 1. For example, $S = s_1 s_2 \ldots s_n \in \Sigma^n$. The length of a string $S$ is denoted by $|S|$.

The properties of strings derive from the properties of the alphabet $\Sigma$. In this dissertation, we assume that the alphabet is (or can, in constant time per character, be converted to) the set $\{1, 2, \ldots, |\Sigma|\}$ of integers. In general, the size $|\Sigma|$ is not assumed to be constant or small. However, we do assume that a character fits into one computer word, i.e., operations on characters execute in constant time. The size $|\Sigma|$ will be explicitly included in time and space requirements when necessary. For example, the space requirement of a string $S \in \Sigma^*$ is $\Theta(|S| \log |\Sigma|)$ bits.

The order of the alphabet generalizes naturally to the lexicographic order of strings.

**Definition 2.5** Let $S = s_1 s_2 \ldots s_m$ and $T = t_1 t_2 \ldots t_n$ be strings over $\Sigma$. Let $k$ be the largest integer such that $s_i = t_i$ for all $1 \leq i \leq k$. In the *lexicographic order* of $\Sigma^*$, $S \leq T$ if $k = m$, or $k < m$, $k < n$ and $s_{k+1} < t_{k+1}$.

### 2.2.2 Ranges

**Definition 2.6** A (one-dimensional) *universe* $U$ is an ordered set. The elements of $U$ are called *points*. Let $i$ and $j$ be two points of $U$. The pair $[i, j]$ is a *(closed) range* of $U$ and its interpretation is the set $\{x \in U \mid i \leq x \leq j\}$. The pairs $[i, \infty[$ and $]-\infty, j]$ are *semi-infinite ranges* of $U$ and their interpretations are the sets $\{x \in U \mid i \leq x\}$ and $\{x \in U \mid x \leq j\}$, respectively. A range is *empty* if its interpretation is empty.

**Remark 2.7** There are also *open* and *semiopen* ranges. For example:

$$\begin{array}{rll} ]i,j[ \text{ with interpretation } \{x \in U \mid i < x < j\} & \text{(open)} \\ [i,j[ \text{ with interpretation } \{x \in U \mid i \leq x < j\} & \text{(semiopen)} \\ ]i,\infty[ \text{ with interpretation } \{x \in U \mid i < x\} & \text{(semi-infinite)} \end{array}$$

For simplicity, we will concentrate on closed ranges, but all that follows generalizes for the other types of ranges. The semi-infinite ranges will have special significance later.

A range has two roles: its representation as a pair of points and its interpretation as a set of points. The pair role is the 'true identity'; it defines the interpretation as a set. (The pair could be defined as the minimum and maximum of the set, except for empty ranges.) However, we will not define all necessary set operations and relations for the pair representation. In set contexts, a range behaves as if it was a set. The only possibility for confusion is the containment relation, which is a natural set relation but also defined for pairs of points in Definition 2.8; see the comments after the definition.

There is no natural full order for ranges but there are the following two partial orders.

**Definition 2.8** A range $[i,j]$ is *contained* by a range $[i',j']$, denoted by $[i,j] \subseteq [i',j']$, if $i' \leq i$ and $j \leq j'$. There are two degrees of inequality:

$$\begin{array}{lll} [i,j] \subset [i',j'] & \text{if} & [i,j] \subseteq [i',j'] \text{ and } [i,j] \neq [i',j']. \\ [i,j] \Subset [i',j'] & \text{if} & [i,j] \subseteq [i',j'],\ i \neq i' \text{ and } j \neq j'. \end{array}$$

In the last case, we say that $[i,j]$ is *nested* within $[i',j']$.

The range containment is mostly consistent with the set containment. There are two reasons why the set containment is not enough. First, the nesting relation is not defined for sets, although it could be defined using minimum and maximum. Second, and more importantly, the above definition may differ from the set containment, when empty ranges are involved.

**Definition 2.9** A range $[i,j]$ *precedes* a range $[i',j']$, denoted by $[i,j] \leq [i',j']$, if $i \leq i'$ and $j \leq j'$. There are two degrees of inequality:

$$\begin{array}{lll} [i,j] < [i',j'] & \text{if} & [i,j] \leq [i',j'] \text{ and } [i,j] \neq [i',j']. \\ [i,j] \ll [i',j'] & \text{if} & [i,j] \leq [i',j'],\ i \neq i' \text{ and } j \neq j'. \end{array}$$

Two ranges $[i,j]$ and $[i',j']$ are *comparable* in $\subseteq$ if $[i,j] \subseteq [i',j']$ or $[i',j'] \subseteq [i,j]$. Otherwise, $[i,j]$ and $[i',j']$ are *incomparable* in $\subseteq$. The comparability in $\leq$ is defined equivalently.

**Example 2.10** $[3,4] \Subset [2,5] \subset [2,7]$, but $[1,3]$ and $[2,5]$ are incomparable in $\subseteq$. $[1,3] \ll [2,5] < [2,7]$, but $[2,5]$ and $[3,4]$ are incomparable in $\leq$.

The two partial orders are related by the following lemma.

**Lemma 2.11 (Orthogonality)** *Two ranges $[i,j]$ and $[i',j']$ are always comparable in either $\subseteq$ or $\leq$. The ranges are comparable in both $\subseteq$ and $\leq$ iff $i = i'$ or $j = j'$.*

We say that $\subseteq$ and $\leq$ are *orthogonal.*

### 2.2.3 Substrings and intervals

A substring of a string has a dual role. It is a string but also a range of positions within another string. For the purposes of this dissertation, we need to make a clear distinction between the roles. We will use the term substring only for the string role and the term *interval* for the position role. The precise definitions are below.

**Definition 2.12** Let $T = t_1 t_2 \ldots t_n$ be a string. A range $[i,j]$ of integers is an *interval* of $T$ if $1 \leq i \leq j+1 \leq n+1$. The interval is *empty* if $j = i-1$.

The notation for interval $[i,i]$ can be shortened to $[i]$. In what follows, a single capital letter is often used for denoting an interval. The starting and ending positions of an interval, say, $I$ are then denoted by $\overleftarrow{I}$ and $\overrightarrow{I}$, respectively, i.e., $I = [\overleftarrow{I}, \overrightarrow{I}]$.

**Definition 2.13** Let $T = t_1 t_2 \ldots t_n$ be a string and $I$ its interval. The *substring* $T(I)$ is the string $t_{\overleftarrow{I}} t_{\overleftarrow{I}+1} \ldots t_{\overrightarrow{I}}$.

When the interval is represented using brackets, the substring expression can be shortened by removing the parenthesis. For example $T([i,j])$ is shortened to $T[i,j]$. Note that then $T[i] = t_i$.

**Example 2.14** Let $T = \mathtt{ABCDEFG}$. Then $T[2,4] = \mathtt{BCD}$, $T[5] = \mathtt{E}$ and $T[4,3] = \epsilon$, but $T[5,2]$ and $T[6,9]$ are undefined, because $[5,2]$ and $[6,9]$ are not intervals of $T$.

Prefixes and suffixes are special types of substrings.

**Definition 2.15** Let $T = t_1 t_2 \ldots t_n$ be a string. A string $S$ is a *prefix* of $T$ if $S = T[1, j]$ for some $0 \leq j \leq n$. A string $S$ is a *suffix* of $T$ if $S = T[i, n]$ for some $1 \leq i \leq n + 1$.

### 2.2.4 Two-dimensional ranges

Finally, we generalize ranges into two dimensions. The duality of representation with bounding values and interpretation as a set is inherited from the one-dimensional case.

**Definition 2.16** Let $U_1$ and $U_2$ be one-dimensional universes (ordered sets). The Cartesian product $U = U_1 \times U_2$ is a *two-dimensional universe* and $U_1$ and $U_2$ are the first and the second *dimension* of $U$, respectively. An element $i = (i_1, i_2)$ of $U$ is called a *point* of $U$ and $i_1$ and $i_2$ are the first and the second *coordinate* of $i$, respectively. Let $R_1$ be a range of $U_1$ and $R_2$ a range of $U_2$. Then $R = R_1 \times R_2$ is a *two-dimensional range* of $U$ and its interpretation is the Cartesian product of the interpretations of $R_1$ and $R_2$. If either $R_1$ or $R_2$ is a semi-infinite range, $R$ is a *semi-infinite two-dimensional range* of $U$. If both $R_1$ and $R_2$ are semi-infinite ranges, $R$ is a *doubly semi-infinite two-dimensional range* of $U$.

## 2.3 Basic problems

Next we describe some basic problems and their relations to each other. The first problem, pattern matching, is the main subject of this dissertation. The other problems appear as subproblems in our algorithms.

All the problems have a similar form: A large collection of data is *preprocessed* to obtain a *data structure*, sometimes known as an index or a dictionary, that supports *queries*. The main characteristics of a solution are the space requirement of the data structure and the query time. Less important are the preprocessing time and space, since preprocessing needs to be done only once. This kind of problems are known as *static*, *offline* or *indexed*. Most of the problems also have a well-studied dynamic, online or unindexed version, which does not have the (complete) data available for separate preprocessing. In particular, a *dynamic* version calls for a data structure that supports operations that change the data (usually insertions and deletions) in addition to queries. In this dissertation, we will not consider the dynamic, online or unindexed versions.

### 2.3.1 Pattern matching

The pattern matching problem is the target of the methods in this dissertation. Let us start by defining the concept of a pattern.

**Definition 2.17** A *pattern* is a predicate for strings. A string $S$ is said to *match* a pattern $P$ if $P(S)$ is true.

**Remark 2.18** This definition is much more general than what is usually meant with the term pattern, but it is adequate for the purposes of this dissertation. In the more normal use, the term pattern is used only for special kinds of predicates with a description consisting of *symbols*. A string $S$ matches the pattern $P$ if there exists a (partial) matching between the characters of $S$ and the symbols of $P$ satisfying given conditions. The conditions should be verifiable quickly; the difficulty (if any) is finding a satisfying matching.

The most common pattern type is the string pattern.

**Definition 2.19** A pattern $P$ is a *string pattern* if there exists a unique string, also denoted by $P$, such that a string $S$ matches the pattern $P$ if and only if $S$ equals the string $P$. A *q-gram pattern* is a string pattern of length $q$.

Another much studied pattern type is the approximate string pattern.

**Definition 2.20** Let $P$ and $Q$ be two strings. The *edit distance* $\delta(P, Q)$ between $P$ and $Q$ is the smallest number of single-character insertions, deletions and substitutions needed to change $P$ into $Q$.

**Definition 2.21** Let $P$ be a string and $d$ a non-negative integer. An *approximate string pattern* $(P, d)$ is matched by a string $S$ if $\delta(S, P) \leq d$.

There are several other types of patterns, e.g., approximate string patterns with a different distance measure, regular expressions, and episodes [DFG+97]. In this dissertation, the main focus is on string patterns. However, many aspects of the methods developed here are independent of the particular type of pattern, so we will use the general concept of pattern in Definition 2.17 whenever possible.

The simplest type of a pattern matching problem is to find whether a string $S$ matches a pattern $P$ (and possibly to find the actual matching, too). A more difficult type of pattern matching is to find substrings of a long string that match the pattern. The main problem studied in this dissertation is the indexed version of the latter problem defined as follows.

**Problem 2.22 (Pattern matching)** Let $T$ be a string called the *text* and let $P$ be a pattern. An *occurrence* of $P$ in $T$ is an interval $I$ of $T$ such that $T(I)$ matches $P$. The *pattern matching query* asks for all occurrences of $P$ in $T$. Let $\mathcal{P}$ be a class of patterns. The *(indexed) pattern matching problem* is to store the text $T$ in a data structure called the *(text) index* in such a way that pattern matching queries can be answered quickly for any pattern in the class $\mathcal{P}$.

The specific pattern matching problems encountered in this dissertation are *string matching* ($\mathcal{P}$ is the class of string patterns), *$q$-gram matching* ($\mathcal{P}$ is the class of $q$-gram patterns), and *approximate string matching* ($\mathcal{P}$ is the class of approximate string patterns).

### 2.3.2 Range searching

The range search problem is a much studied problem in computational geometry. Together with its two-dimensional generalization, it is an important subproblem in the new string matching algorithms developed in this dissertation.

**Problem 2.23 (Range searching)** Let $U$ be a universe, $V$ a (multi)set of points of $U$, and $R$ a range of $U$. The *range query* asks for the (multi)set $V \cap R$, i.e., the points of $V$ that are in the range $R$. The *(static) range search problem* is to store $V$ in such a way that range queries can be answered quickly for any range $R$.

There is also another, more direct connection between range searching and string matching. The connecting link is the prefix matching problem.

**Problem 2.24 (Prefix matching)** Let $V$ be a set of strings and $Q$ a string in $\Sigma^*$. The *prefix matching query* asks for the strings in $V$ that have $Q$ as a prefix. The *(static) prefix matching problem* is to store $V$ in such a way that prefix matching queries can be answered quickly for any string $Q$.

Prefix matching is a special case of range searching in the universe $\Sigma^*$, because the set of strings with the prefix $Q$ forms a (semi-open) range of $\Sigma^*$ in the lexicographic ordering. On the other hand, the string matching problem can be reduced to prefix matching. Let $V_T$ be the set of suffixes of a text $T$. Then the prefix matching query with $V = V_T$ and $Q = P$ finds the suffixes of $T$ that contain the pattern $P$ as a prefix. The starting positions of the suffixes are exactly the starting positions of the occurrences of $P$ in $T$. Thus, any solution to the range search problem in the universe of strings can be used for solving the string matching problem. Note that this reduction does not extend to, e.g., approximate string matching, because an answer to an approximate string matching query cannot be represented as a single range.

### 2.3.3 Ranking

Ranking is a basic one-dimensional search operation. Many one-dimensional queries, including range queries, are easy to implement using ranking.

**Problem 2.25 (Ranking)** Let $U$ be the universe. Let $V$ be a finite multiset of points in $U$ and $q \in U$. The minimum and maximum ranks of $q$ with respect to $V$ are

$$\begin{aligned}\text{minrank}_V(q) &= |\{v \in V \mid v < q\}| + 1 \\ \text{maxrank}_V(q) &= |\{v \in V \mid v \leq q\}| + 1\end{aligned}$$

The *ranking query* asks for $\text{minrank}_V(q)$ and $\text{maxrank}_V(q)$. The *(static) ranking problem* is to store $V$ in such a way that ranking queries can be answered quickly for any $q \in U$. The *access query* asks whether $q$ is in $V$, and if it is, it asks for $\text{minrank}_V(q)$ and $\text{maxrank}_V(q)$. The *(static) access problem* is to store $V$ in such a way that access queries can be answered quickly for any $q \in U$.

Ranking answers the question: What would be the rank of $q$ if inserted into $V$? If $V$ already contains one or more instances of $q$, the rank could be anything from a range of values. The minimum and maximum ranks give the extremes. If $V$ does not contain $q$, the minimum and maximum ranks are equal. If $V$ is not a multiset, the minimum and maximum ranks differ by at most one, and computing just one rank is sufficient in most cases.

**Remark 2.26** Instead of returning a rank, we could return a pointer. The pointer version of ranking is neighbor search, which returns a pointer to the predecessor and/or successor of a query point. The pointer version of the access problem is known as the dictionary problem. The first part of the access query, finding whether $q$ is in $V$, is also known as the membership query. In the static setting (and thus in this dissertation) the rank version and the pointer version are essentially the same problem. In dynamic setting, however, a single insertion or deletion can change the ranks of all points, which adds an extra degree of difficulty to ranking. On the other hand, ranks cannot be replaced with pointers, e.g., in Lemma 2.27.

The use of the ranking operations is demonstrated in the following lemma.

**Lemma 2.27** *Let the multiset $V$ consist of the elements $v_1 \leq v_2 \leq \ldots \leq v_N$ and let $q, l, h \in U$. Then $v_i = q$ for $minrank_V(q) \leq i \leq maxrank_V(q) - 1$. If $q \notin V$, the nearest neighbors of $q$ in $V$ are $v_{minrank_V(q)-1} < q$ and $v_{minrank_V(q)} > q$. Furthermore,*

$$\begin{aligned}
V \cap [l,h] &= \{v_i \in V \mid i \in [minrank_V(l), maxrank_V(h) - 1]\}, \\
V \cap [l,h[ &= \{v_i \in V \mid i \in [minrank_V(l), minrank_V(h) - 1]\}, \\
V \cap ]l,h] &= \{v_i \in V \mid i \in [maxrank_V(l), maxrank_V(h) - 1]\}, \\
V \cap ]l,h[ &= \{v_i \in V \mid i \in [maxrank_V(l), minrank_V(h) - 1]\} \text{ and} \\
V \cap [l,\infty[ &= \{v_i \in V \mid i \in [minrank_V(l), N]\} \text{ and} \\
V \cap ]{-\infty},h] &= \{v_i \in V \mid i \in [1, maxrank_V(h) - 1]\}.
\end{aligned}$$

The lemma shows how ranking can be used for transforming a range in the universe $U$ to a range in the *rank space* $\{1, 2, \ldots, N\}$.[1] After the transformation, answering the range query is easy. Thus, range searching, and consequently string matching, too, can be reduced to ranking.

### 2.3.4 Two-dimensional range searching

Finally, we consider two-dimensional generalizations of range searching.

**Problem 2.28 (Two-dimensional range searching)** Let $U$ be a two-dimensional universe, $V$ a (multi)set of points of $U$, and $R$ a two-dimensional range of $U$. The *two-dimensional range query* asks for the (multi)set $V \cap R$, i.e., the points of $V$ that are in the range $R$. The *(static) two-dimensional range search problem* is to store $V$ in such a way that range queries can be answered quickly for any range $R$. If $R$ is restricted to be a (doubly) semi-infinite two-dimensional range, the query and the problem are also called (doubly) semi-infinite.

**Remark 2.29** The above type of ranges are often called orthogonal or rectilinear to distinguish them from other types of ranges. Also the type of query is often called range reporting in contrast to range counting and other types of queries. See, e.g., [Aga97] for more details.

[1] The values 0 and $N+1$ can also appear in the transformed ranges but only in the empty ranges $[1,0]$ and $[N+1,N]$, which are easily handled as special cases.

The one-dimensional components of a two-dimensional range can be any combination of closed, open, semi-open or semi-infinite ranges. We define the semi-infinite form of the problem separately, because it is significantly easier than the general problem. Semi-infinite range searching is also known as three-sided range searching, and doubly semi-infinite range searching as two-sided range searching and dominance searching.

Lemma 2.27 showed how, given a multiset $V$ of a one-dimensional universe $U$, any one-dimensional range of $U$ can be transformed into a range of the rank space $\{1, 2, \ldots, |V|\}$. In the two-dimensional case, we can do such a transformation separately in each dimension. This transforms a two-dimensional range $R$ of a universe $U$ into a two-dimensional range $R'$ of the two-dimensional rank space $U' = \{1, 2, \ldots, |V|\} \times \{1, 2, \ldots, |V|\}$. The point set $V \subseteq U$ is transformed into $V' \subseteq U'$ by replacing each point with the pair of its ranks in the two dimensions. If more than one points have the same coordinate value in one dimension, then, say, the coordinates in the other dimension can be used for deciding the ordering of the points; what is important is that no two points get the same rank. Now the range query $R' \cap V'$ gives exactly the points of $V'$ that replaced the points in $R \cap V$.

The transformation to the rank space has many benefits for two-dimensional range searching. First, the universe may be reduced significantly. Second, in the transformed point set $V'$, each integer $i \in [1, |V|]$ appears exactly once as a first coordinate and exactly once as a second coordinate.[2] This simplifies some algorithms because there are no duplicate values and because a particular value can be assumed to be present. Third, any type of range is transformed into a closed range (see Lemma 2.27). However, the one-dimensional ranges starting with 1 or ending with $|V|$ can be considered to be semi-infinite.

After the transformation to the rank space, we are left with solving the following problem.

**Problem 2.30 (Two-dimensional range searching in rank space)** Let $V$ be a set of $N$ points in the two-dimensional universe $[1, N]^2 = [1, N] \times [1, N]$ such that each $i \in [1, N]$ appears exactly once as a first coordinate and exactly once as a second coordinate in $V$. Let $R = R_1 \times R_2 = [l_1, h_1] \times [l_2, h_2]$ be a range of $[1, N]^2$. The two-dimensional range query asks for the points $V \cap R$. The two-dimensional range search problem is to store $V$ in such a way that the range query can be answered quickly for any range. If $l_1 = 1$, $l_2 = 1$, $h_1 = N$ or $h_2 = N$, the range $R$ and the corresponding query and problem are called semi-infinite.

[2] This makes $V'$ a *permutation* of the set $[1, N]$.

Finally, consider the following one-dimensional problem.

**Problem 2.31 (One-dimensional range containment)** Let $U$ be a one-dimensional universe, $V$ a set of ranges of $U$, and $R$ a range of $U$. The *range containment query* asks for the set $\{I \in V \mid R \subseteq I\}$, i.e., ranges in $V$ that contain $R$. The *(static) range containment problem* is to store the set $V$ in such a way that range containment queries can be answered quickly.

A one-dimensional range can be seen as a two-dimensional point with the end points as its coordinates. Thus, the test $R \subseteq I$ is equivalent to the test $(\overleftarrow{I}, \overrightarrow{I}) \in \,]-\infty, \overleftarrow{R}] \times [\overrightarrow{R}, \infty[$. In other words, a solution to the doubly semi-infinite two-dimensional range search problem is also a solution to the one-dimensional range containment problem.

# Chapter 3

# Solutions to basic problems

In this chapter, we review some current approaches to the basic problems of ranking, one- and two-dimensional range searching, and string matching. As was noted in the previous chapter, these problems are closely related with ranking being the fundamental problem. This chapter serves two purposes, to give solutions to subproblems that rise in repetition-based indexing, and to establish the properties of previously known methods for string matching.

The focus of this chapter is on establishing asymptotic worst-case bounds on the space requirement of the data structures and the time requirement for answering queries. To keep things simple, we will mostly ignore preprocessing, dynamic insertion and deletion, average case analysis, and practical implementation. Furthermore, the emphasis is more on small space requirement than fast query time.

Due to the somewhat nonstandard model of computation, we describe and analyze the solutions in detail instead of just referring to the literature. This also allows us to provide a more unified view to the different methods.

There are also some new results. Most of the string-related results are improvements over previous results because bit-packing allows faster string comparison. Also the use of $y$-fast tries in implementing the nodes of string tries and suffix trees is new. For two-dimensional range searching, we give a variation of a data structure by Overmars [Ove88] with new possibilities for space–time tradeoffs.

## 3.1 Ranking integers

We start with the ranking problem (Problem 2.25) in small, finite universes. Let us assume that the universe is the range $[1, |U|]$ of integers, where $|U|$ is small enough to fit into a constant number of computer words.

The elements of the set $V$ are called *$V$-points*. For simplicity, we assume that the set $V$ is a simple set, not a multiset. A multiset $V'$ can be handled by providing a data structure for ranking with respect to a set $V$, which is $V'$ with duplicates removed, and a lookup table of $O(|V|\log|V'|)$ bits to map a rank in $V$ to a rank in $V'$.

By the above assumptions, the parameters of the problem are related by $|V| \leq |U| = 2^{O(w)}$. We will not assume any upper bound on the word size $w$, because integer ranking with small sets or small universes is a frequent subproblem in tasks with much larger space requirements. In such cases, the word size is determined by the larger task.

### 3.1.1 Constant-time techniques

We start by describing some data structures with a constant query time.

The simplest constant-time data structure for ranking is the lookup table of $O(|U|\log|V|)$ bits that stores the ranks of every point in the universe. Another basic technique is based on representing $V$ as a bitmap of size $|U|$. The ranks of a query point $q$ can be computed by counting the number of 1 bits before the $q$th bit. We can count $b$ bits at a time with a lookup table of $2^b$ entries and $O(\log b)$ bits per entry. These two techniques can be combined as follows.

**Theorem 3.1 (Bucketed bitmap)** *A set $V$ on a universe $U = [1, |U|]$ can be stored in $O(|U|)$ bits so that ranking queries can be answered in $O(1)$ time.*

**Proof** Divide the universe $U$ into buckets of size $b = \log|V| - \log\log|V|$. The ranks at the bucket boundaries are stored into a lookup table size $O((|U|/b)\log|V|) = O(|U|)$ bits. The ranks within a bucket are computed by counting bits in a bitmap of $b$ bits. In constant time we can extract the relevant bits and count them by using a lookup table of size $O(2^b \log b) = O(|V|)$ bits. All the data structures, the lookup table for bucket boundary ranks, the bitmaps, and the lookup table for bit counting, fit into $O(|U|)$ bits. $\square$

The bucketed bitmap is the first example of the bucket technique for constructing a hybrid data structure that is better than its constituent data structures.

The set $V$ defines a partition of the universe $U$ into two subsets, $V$ and $U \setminus V$. The idea of ranking can be generalized to partitions of more subsets. The following result generalizes the bucketed bitmap method for general partitions. The method is, in its essence, described in [Cha88] to solve a subproblem (see Section 3.6.1).

**Lemma 3.2** *Let $V_1, V_2, \ldots, V_m$ be a partition of the universe $U$. The partition can be stored in $O(|U| \log m)$ bits so that, for any $q \in U$, the set $V_i$ that contains $q$ and the ranks of $q$ in $V_i$ can be computed in $O(1)$ time.*

**Proof** Let $A$ be a table of $|U|$ entries, where the $p$th entry $A_p$ is $j$ if $p \in V_j$. Then $i = A_q$, and, by counting the number of occurrences of $i$ in $A$ up to the $q$th entry, we get the ranks of $q$ in $V_i$. To speed up the counting, we use two-level bucketing.

Divide $U$ into $|U|/(m \log |U|)$ buckets of size $O(m \log |U|)$; we call these the *major buckets*. For each $j \in [1, m]$, store the ranks of the first point $p$ in each major bucket with respect to $V_j$, i.e., the number of occurrences of $j$ in $A$ up to the position $p$, to a table $B_1$. Divide each major bucket further into $O(m \log |U|/b)$ smaller buckets, called *minor buckets*, of size $O(b)$. For each $j \in [1, m]$ and each first point $p'$ of a minor bucket, store the number of occurrences of $j$ in $A$ between the position $p'$ and the previous major bucket boundary into a table $B_2$.

The space requirement of the table $A$ is $O(|U| \log m)$ bits, the table $B_1$ $O(|U|)$ bits, and the table $B_2$ $O((|U|/b) m \log(m \log |U|)) = O(|U| m(\log m + \log\log |U|)/b)$ bits. With the tables $B_1$ and $B_2$, we can compute the number of occurrences of $i$ in $A$ up to the starting position $p'$ of the minor bucket that contains $q$. To get the answer, we still need to compute $c_q$, the number of occurrences of $i$ in $A$ between the positions $p'$ and $q$.

If $m \geq \log\log |U|$, let $b = m^2$ and store $c_q$, for all $q \in U$, to a table $B_3$. The table $B_3$ requires $O(|U| \log b) = O(|U| \log m)$ bits of space. With $b = m^2 = \Omega(m(\log m + \log\log |U|))$, the table $B_2$ requires $O(|U|)$ bits.

If $m < \log\log |U|$, let $b = \log_m |U| - \log_m \log\log |U|$. Now a whole minor bucket of $A$ fits into $O(b \log m) = O(\log |U|) = O(w)$ bits. We can extract, in constant time, the section of $A$ between positions $p'$ and $q$ with $q$ being the last position included. Then, we use a lookup table $C$ to find how many times the last entry (which is $i$) occurs in the section. The space requirement of the table $C$ is $O(m^b \log b) = O(|U|)$ bits. With $b = \Omega(\log_m |U|) = \Omega(m(\log m + \log\log |U|))$, the space requirement of the table $B_2$ is $O(|U|)$ bits. □

Another well-known constant-time technique is hashing. Hashing cannot be used directly to implement ranking, but it is an efficient solution to the access problem. There are a wide variety of hashing methods with different properties (see e.g. [Knu98]). A hash function $h$ with the property that

$h(x)$ is different for each $x \in V$ is called a *perfect hash function* for the set $V$. Czech et al. [CHM97] give a comprehensive review of perfect hashing methods. The following result is due to Fredman et al. [FKS84].

**Lemma 3.3 (Perfect hashing)** *A set $V$ on a universe $U = [1, |U|]$ can be stored in $O(|V| \log |U|)$ bits so that access queries can be answered in $O(1)$ time.*

**Proof** Let $p$ be a prime larger than $|U|$ (but not by much). For a hash table of size $s$, the hash function is chosen from the class $h_a^s(x) = (ax \bmod p) \bmod s$, $a \in [1, p-1]$. Let $V$ be an arbitrary set of points. Fredman et al. show that there exists $a \in [1, p-1]$ such that $h_a^{|V|^2}$ is a perfect hash function for $V$. Furthermore, they show that there exists $a \in [1, p-1]$ such that the hash function $h_a^{|V|}$ satisfies

$$\sum_{j=0}^{|V|-1} \left|\{x \in V \mid h_a^{|V|}(x) = j\}\right|^2 < 3|V|.$$

We omit the proofs of these results here.

The data structure consists of a hash table of size $|V|$ and, for each bucket containing the points $V_j$, a second level hash table of size $|V_j|^2$. By the above results, the hash functions can be chosen to make the total number of hash table entries $O(|V|)$, and all the second level hash functions perfect. □

**Remark 3.4** Fredman et al. give a deterministic construction algorithm that works in $O(|V|^3 \log |U|)$ time, and a randomized construction algorithm with expected time $O(|V|)$. Raman [Ram96] suggests using a different hash function in the same two-level scheme. Raman's hash function does not need to know a prime larger than the universe, does not need integer division (but does need multiplication) and can be constructed in $O(|V|^2 \log |U|)$ deterministic time.

Another method worth mentioning is *fusion*, the constant-time ranking method used in fusion trees [FW93]. For $|V|^6 \leq w$, a rank query can be answered in constant time using a data structure of $O(w + |V| \log |U|)$ bits [And96] and a complicated sequence of bit-parallel operations. Variations are described in [Hag98, AMT99]. However, we do not use this method.

### 3.1.2 Trees and tries

The rest of the data structures work by the divide-and-conquer principle: repeatedly divide the universe into smaller ranges of roughly equal "weight" until the solution is found. There are two basic types of such data structures for ranking: trees and tries. With trees the weight of a subrange is the number of $V$-points in it; with tries the weight is the length of the subrange.

A simple but effective tree structure is the balanced binary search tree. In a static setting, it can be implemented as an ordered array of the $V$-points searched by a binary search.

**Theorem 3.5 (Binary tree)** *A set $V$ on a universe $U = [1, |U|]$ can be stored in $O(|V| \log |U|)$ bits so that ranking queries can be answered in $O(\log |V|)$ time.*

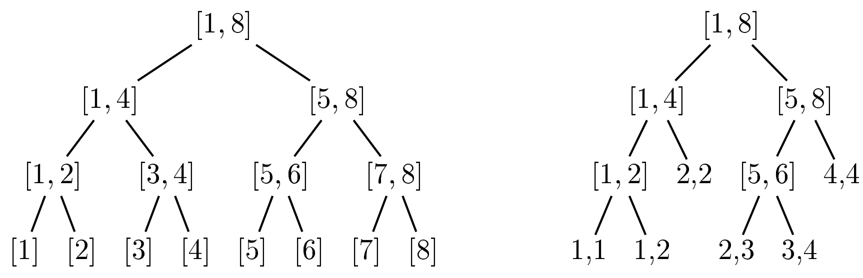
We will not need other tree methods, but worth mentioning are the fusion tree [FW93] and the exponential search tree by Andersson [And96]. Both are higher degree trees and use fusion to implement the nodes. The latter also uses some of the trie techniques presented below to achieve $O(\sqrt{\log |V|})$ query time in space of $O(|V| \log |U|)$ bits.

Next we describe trie structures. Let $Tree_a(U)$ denote the tree with *arity* $a$, i.e., with all internal nodes having $a$ children. (This may require increasing $|U|$ by at most factor $a$.) The leaves of the tree represent the points of $U$ and the internal nodes represent ranges of $U$. An example is shown in Figure 3.1(a). The tree can be used for ranking by storing, for every $q \in U$, the ranks of $q$ in the leaf representing $q$.

Now, consider a subtree that contains no $V$-points. All the ranks in the subtree are the same. Thus the subtree can be replaced with a single leaf node storing the common ranks. By collapsing all the empty subtrees, we get the trie $Trie_a(U, V)$. An example is shown in Figure 3.1(b).

The advantage of the trie $Trie_a(U, V)$ over the binary tree of $V$ is its regular structure. The structure of the tree $Tree_a(U)$ is perfectly regular in the sense that the nodes can be stored in a table so that, for any $q \in U$ and any level $\ell$ of the tree, the location of the node on the level $\ell$ whose range contains $q$ can be computed in constant time. The trie $Trie_a(U, V)$ has the same structure except that some nodes are missing. Thus the trie can be stored similarly in a sparse table implemented, e.g., with hashing. The following result due to Willard [Wil83] takes advantage of this possibility.

**Lemma 3.6 ($x$-fast trie)** *A set $V$ on a universe $U = [1, |U|]$ can be stored in $O(|V| \log^2 |U|)$ bits so that ranking queries can be answered in $O(\log \log |U|)$ time.*




(a) $Tree_2([1, 8])$ (b) $Trie_2([1, 8], \{2, 5, 6\})$

Figure 3.1: (a) $Tree_2([1, 8])$ with each node marked with the range it represents. (b) $Trie_2([1, 8], \{2, 5, 6\})$ with each leaf marked with the minrank and maxrank values it stores.

**Proof** Store $V$ into the trie $Trie_2(U, V)$. The nodes of the trie are stored in a hash table. The hash key of a node is its location in $Tree_2(U)$. Each entry of the hash table stores the hash key and, if the node is a leaf, the corresponding ranks. Every node fits in $O(\log |U|)$ bits; thus the space requirement is $O(|V| \log^2 |U|)$ bits.

Consider now a ranking query with $q \in U$. Let $\pi$ be the path from the root to the leaf representing $q$ in the complete tree of $U$. Using the hash table, we can, in $O(1)$ time, find out whether a given node on $\pi$ exists in the trie. By a binary search on the path $\pi$, we can locate the leaf of the trie whose range contains $q$ in $O(\log \log |U|)$ time. □

The above trie requires more space than the binary tree. However, by combining the trie with trees using the bucket technique we can reduce the space requirement. The resulting data structure is called the $y$-fast trie [Wil83].

**Theorem 3.7 ($y$-fast trie)** *A set $V$ on a universe $U = [1, |U|]$ can be stored in $O(|V| \log |U|)$ bits so that ranking queries can be answered in $O(\log \log |U|)$ time.*

**Proof** Divide $V$ into buckets of size $\log |U|$ and store each bucket in a binary tree. The $O(|V|/\log |U|)$ bucket separators are stored in an $x$-fast trie (Lemma 3.6). The binary trees need $O(|V| \log |U|)$ bits and the $x$-fast trie $O((|V|/\log |U|) \log^2 |U|) = O(|V| \log |U|)$ bits of space. A query consists

Table 3.1: Data structures for ranking integers.

| data structure | query time | space |
|---|---|---|
| bucketed bitmap (Theorem 3.1) | $O(1)$ | $O(\lvert U\rvert)$ |
| binary tree (Theorem 3.5) | $O(\log\lvert V\rvert)$ | $O(\lvert V\rvert\log\lvert U\rvert)$ |
| $y$-fast trie (Theorem 3.7) | $O(\log\log\lvert U\rvert)$ | $O(\lvert V\rvert\log\lvert U\rvert)$ |

of a query in the $x$-fast trie followed by a query in a binary tree, both of which require $O(\log\log|U|)$ time. □

Another nonobvious implementation of a trie is given in [Wil84] (see also [FW93]).

### 3.1.3 Summary

Table 3.1 summarizes the properties of the integer ranking data structures we have described. Storing just the $V$-points needs $\Omega(|V|\log|U|)$ bits; thus the space requirement of binary trees and $y$-fast tries is optimal in the general case. Note, however, that, if the universe is small compared to the set $V$, the space requirement of the bucketed bitmap can be $o(|V|\log|U|)$.

## 3.2 Ranking strings

In this section, we consider the ranking problem when the universe is the set $\Sigma^*$ of strings, where $\Sigma = \{1, \ldots, |\Sigma|\}$ and $|\Sigma|$ is small enough to fit into a constant number of computer words.

As before, we assume that the set $V$ is a simple set, not a multiset. The elements of $V$ are called $V$-strings.

### 3.2.1 Representing strings

Before presenting the data structures for ranking, we need to specify some details of string representation.

We assume that characters are stored as strings of $\lceil\log(|\Sigma|+1)\rceil$ bits in the natural way. The bit string of all zeros is not used. A string is stored as the concatenation of the character bit strings tightly packed into words. Possibly unused part of the last word is filled with zeros. Thus a string $S$ can be seen as a string of $|S|\lceil\log(|\Sigma|+1)\rceil$ bits or a string of $\lceil|S|\lceil\log(|\Sigma|+1)\rceil/w\rceil$ words. The lexicographic ordering is preserved in either representation.

Thus, using the string of words representation, comparisons are faster than usual by the factor $\Theta(w/\log|\Sigma|)$.

Another string representation matter is storing the set $V$. Let $||V||$ be the total length of the $V$-strings. In general, we need $\Theta(||V||\log|\Sigma|)$ bits to store the set $V$. However, there are relevant special cases, for example the set of all suffixes of a string, where $V$ can be stored in less space. To account for such cases, we define the following storage scheme for the set $V$.

Let $T$ be a common superstring of the $V$-strings. That is, every $V$-string is a substring of $T$. Then each string in $V$ can be represented with two pointers to $T$. The total space requirement of this representation is $O(|V|\log|T| + |T|\log|\Sigma|)$ bits. We can assume that the length of $T$ is at most $||V||$ but it can be much smaller. For example, if $V$ is the set of all suffixes of $T$, then $|T| = |V| = \Theta(\sqrt{||V||})$. Note that any bit, character or word of a $V$-string can still be accessed in constant time with bitwise shift and Boolean operations.

### 3.2.2 Binary tree

The constant-time methods of Section 3.1.1 do not work with strings. The lookup table and the bitmap approaches are impossible, because the universe is infinite. Hashing could be generalized to a $O(|Q|\log|\Sigma|/w)$ time operation for a query string $Q$, but this does not lead to efficient data structures for ranking.

The binary tree, too, is based on a constant-time operation that requires $O(|Q|\log|\Sigma|/w)$ time for a query string $Q$: the operation is comparison. Thus a naive implementation of a balanced binary tree for strings has query time $O(|Q|\log|V|\log|\Sigma|/w)$. Manber and Myers [MM93] have described a technique with which the total comparison time during a query can be reduced to $O(|Q|\log|\Sigma|/w+\log|V|)$. Manber and Myers gave the technique for the implicit binary tree of a binary search; it was generalized for arbitrary binary trees in [Irv95]. This technique is described below. For simplicity, we assume, at first, that no bit-packing of characters is used.

Consider a query in a binary tree at the moment when the query string $Q$ is about to be compared to a string $R$ stored in a node at depth $h$ in the tree. Let $\mathcal{S}_R$ be the set of $h$ $V$-strings stored in the ancestors of the node storing $R$, i.e., the set of strings that $Q$ has already been compared to. Let $S_1, S_2, \ldots, S_h$ be the strings in $\mathcal{S}_R$ in lexicographic order, and let $i$ be the integer such that $S_i < R < S_{i+1}$. The string in $\mathcal{S}_R$ that has the longest common prefix with $R$ must be either $S_i$ or $S_{i+1}$ (in case of a tie choose $S_i$). Assume, without loss of generality, that it is $S_i$, and let $l_R$ be the length of the common prefix. A pointer to $S_i$ and the value $l_R$ are stored with $R$.

Let $S$ be a string in $\mathcal{S}_R$ that has the longest common prefix with $Q$, and let $l_Q$ be the length of the common prefix; $S$ and $l_Q$ have been computed during the search. Since $Q$ is about to be compared to $R$, we must have $S_i < Q < S_{i+1}$. Thus, $S$ is either $S_i$ or $S_{i+1}$ (if it is a tie, choose $S_i$).

The length of the longest common prefix indicates lexicographic closeness of strings. Thus, if $S = S_{i+1}$, then $R$ is closer to $S_i$ while $Q$ is closer to $S_{i+1}$, and therefore, $S_i < R < Q < S_{i+1}$. If $S = S_i$ but $l_Q < l_R$, then $S_i < R < Q$. Symmetrically, if $S = S_i$ but $l_Q > l_R$, then $S_i < Q < R$. In each case, the comparison of $Q$ and $R$ required no character comparisons. In the remaining case, $S = S_i$ and $l_Q = l_R$. Then $Q$ and $R$ have a common prefix of length $l_Q$ and the comparison of $Q$ and $R$ can start from the character $l_Q + 1$. During the character comparisons, a match increases the length of the common prefix and thus the value of $l_Q$ for the following steps, and a mismatch ends the comparison. Therefore, if $d$ is the depth of the binary tree, there are at most $|Q|$ matches and $d$ mismatches during the whole search.

The effect of character packing is that we can compare $\Theta(\log|\Sigma|/w)$ characters at a time and thus each match increases $l_Q$ by $\Theta(\log|\Sigma|/w)$. We have the following theorem.

**Theorem 3.8 (Binary tree for strings)** *A set $V$ of substrings of $T \in \Sigma^*$ can be stored in $O(|V|\log|T| + |T|\log|\Sigma|)$ bits so that the ranks of a query string $Q$ can be computed in $O(|Q|\log|\Sigma|/w + \log|V|)$ time.*

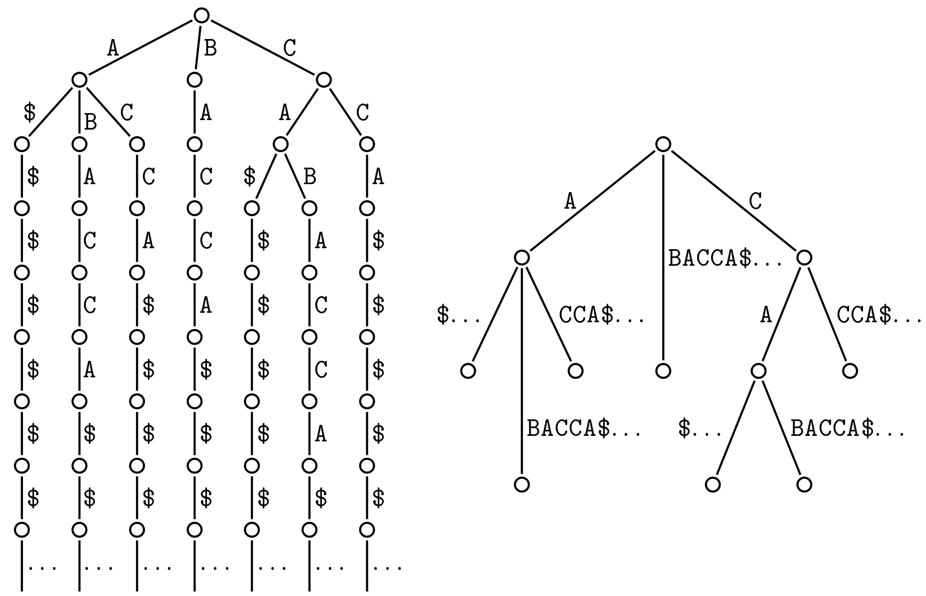
### 3.2.3 Path-compressed tries

Before describing trie structures for strings, we introduce the concept of sistrings [GBY91].

Let $\Sigma_\$$ denote the set $\{\$\} \cup \Sigma$ where the extra character \$ is the first in the ordering. Let $\Sigma_\$^\infty$ be the set of *sistrings* (semi-infinite strings) over $\Sigma_\$$. The set $\Sigma^*$ can be embedded into $\Sigma_\$^\infty$ by appending an infinite sequence of the character \$ to the end of a string $S \in \Sigma^*$. The embedding preserves the lexicographic ordering. The character \$ is represented with the string of $\lceil|\Sigma_\$|\rceil$ zero bits. Thus, sistrings are consistent with the string-of-bits and the string-of-words representations.

We will define tries for the universe $\Sigma_\$^\infty$ instead of $\Sigma^*$. The main reason is to avoid complexities caused by the varying length of strings. In particular, the varying length would be problematic for tries with the arity larger than the alphabet size.

Let us start with the complete (infinite) $|\Sigma_\$|$-ary tree for the universe $\Sigma_\$^\infty$, denoted by $STree_{|\Sigma_\$|}(\Sigma_\$^\infty)$. As before with integers, the nodes of $STree_{|\Sigma_\$|}(\Sigma_\$^\infty)$ represent ranges. We prune the tree by removing (not just




(a) $STrie_4(\{\mathtt{A},\mathtt{B},\mathtt{C}\}_\$^\infty, V)$ (b) $CSTrie_4(\{\mathtt{A},\mathtt{B},\mathtt{C}\}_\$^\infty, V)$

Figure 3.2: (a) trie $STrie_4(\{\mathtt{A},\mathtt{B},\mathtt{C}\}_\$^\infty, V)$, and (b) path-compressed trie $CSTrie_4(\{\mathtt{A},\mathtt{B},\mathtt{C}\}_\$^\infty, V)$, when $V$ is the set of nonempty suffixes of the string CABACCA.

collapsing) the subtrees whose range contains no $V$-strings. The resulting trie is denoted by $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$. An example is shown in Figure 3.2(a).

The nodes of $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$ represent ranges. Because of the arity $|\Sigma_\$|$, each such range has an additional meaning as the range of sistrings with a particular prefix, which we call the *representative prefix* of the node. In Figure 3.2, the representative prefix of a node is the concatenation of the edge labels on the path from the root to the node. The representative prefix is always a prefix of some sistring in the set $V$, and can be represented with pointers to $T$ in $O(\log |T|)$ space.

In principle, the arity of a string trie could be anything. However, the trie with arity $|\Sigma_\$|$ has a couple of advantages. First, the representative prefix is then well-defined. Second, the depth reached by a ranking query is limited by the length of the query string. This may not be true for other arities as the following example shows.

**Example 3.9** Let $\Sigma_\$ = \{\$, \mathtt{A}, \mathtt{B}\}$ be the alphabet and assume that the set

$V$ contains the sistring $S = \texttt{ABBBBBBBBBBB\$\$\$}\ldots$. Consider a ranking query with the query string $Q = \texttt{B\$\$\$}\ldots$. The sistrings $S$ and $Q$ are very close to each other as can be seen when interpreting them as ternary numbers:

$$(0.2)_3 - (0.12222222222)_3 = (0.00000000001)_3.$$

Thus, in a trie of arbitrary arity, one might have to go down to the level where the ranges of nodes are at most $3^{-11}$ wide before $S$ and $Q$ are separated. However, in the ternary trie the separation happens already at depth one because the strings have no common prefix.

The advantages of the arity $|\Sigma_\$|$ can also be achieved with an arity of some power of $|\Sigma_\$|$. By using the string-of-bits representation, the arity can be any power of two up to $2^w$.

The pruning of string tries was defined slightly differently from integer tries: the empty subtrees were removed not just collapsed. The integer trie method is simpler because all internal nodes have a full set of children, but it would take too much space with string tries. To see the effect of not having a full set of children, consider a ranking in $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$. The ranking with a query sistring $Q$ starts at the root and proceeds downwards. At each step, one has to find the appropriate child. Since some children may be missing, finding the child is an access problem in the universe $\Sigma_\$$. The search ends when it either reaches the $\texttt{\$}$-part of $Q$ or runs into a node that was removed in the pruning. In the former case, $Q$ is in $V$ and the ranks are stored in the last node accessed. In the latter case, the last accessed node $u$ has children but the appropriate child $v$ was removed in the pruning. To account for such a case, each internal node stores the ranks of the first and the last $V$-string in its range. The ranks of $Q$ are then found in the preceding or succeeding sibling of $v$ that still exists in $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$. Locating the existing sibling is a ranking problem in the universe $\Sigma_\$$. In summary, one has to do an access query at each step and possibly a ranking query at the end. For the moment, we leave unspecified the *child access and ranking structures* implementing the queries.

Finally, let us consider the size of the trie $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$. The trie is actually infinite but can be made finite by cutting the paths leading to leafs at the point of first $\texttt{\$}$. However, the trie could still have $\Theta(||V||)$ nodes. The number of nodes can be reduced by a technique called *path compression*, which was first introduced by Morrison [Mor68]. We describe this technique next.

A node in $STrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$ has to store three kinds of information: the representative prefix, the child access and ranking structures, and the ranks of the first and the last $V$-strings in the range of the node. Now consider a path segment without branches, i.e., a path segment where all but the last node has just one child (the last node could be either a leaf or an internal node with more than one child). From the information of the last node one can compute all the information of the other nodes on the path: the representative prefixes are prefixes of the last node's representative prefix, no explicit child access and ranking structure is needed when there is just one child, and the ranks are the same for all of the nodes. Thus, the explicit representations of the single-child nodes can be removed. The removed nodes are also called *implicit* nodes. In the resulting compact string trie, denoted by $CSTrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$, each explicit internal node has at least two children. Therefore, the number of explicit nodes is $O(|V|)$. Figure 3.2(b) shows an example of $CSTrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$.

Now we have the following result.

**Lemma 3.10 (Path-compressed trie)** *A set $V$ of substrings of $T \in \Sigma^*$ can be stored in $O(|V|(\log a + \log |T|) + |T| \log |\Sigma|)$ bits so that the ranks of a query string $Q$ can be computed in $O(|Q| \log |\Sigma| / \log a + \log\log a)$ time, where $a$ is a power of two between 2 and $2^w$.*

**Proof** The set $V$ is stored in the trie $CSTrie_a(\Sigma_\$^\infty, V)$. As discussed earlier, the arity $a$ can be any power of two. The child access query is implemented by perfect hashing (Lemma 3.3) and the child ranking query with $y$-fast tries (Theorem 3.7).

The access and ranking structures need $O(|V| \log a)$ bits over the whole trie. The representative prefixes in the nodes need $O(|V| \log |T|)$ bits and the ranks $O(|V| \log |V|)$ bits. Since $|V| < |T|^2$, $\log |V| = O(\log |T|)$. The strings themselves take $O(|V| \log |T| + |T| \log |\Sigma|)$ bits of space.

A ranking query will go at most to depth $O(|Q| \log |\Sigma| / \log a)$ in the trie. Each step needs an access query and a comparison of $\log a$ bits both of which can be done in $O(1)$ time. A possible ranking query at the end needs $O(\log\log a)$ time. □

With the bucket technique we can reduce the space requirement.

**Theorem 3.11 (Hybrid path-compressed trie)** *A set $V$ of substrings of $T \in \Sigma^*$ can be stored in $O(|V| \log |T| + |T| \log |\Sigma|)$ bits so that the ranks of a query string $Q$ can be computed in $O(|Q| \log |\Sigma| / \log a + \log\log a)$ time, where $a$ is a power of two between 2 and $2^w$.*

**Proof** Divide $V$ into buckets of size $\log a$ and store each bucket into a separate binary tree (Theorem 3.8). The $O(|V| / \log a)$ bucket separators are stored in a path-compressed trie (Lemma 3.10). The space requirement for the binary trees is $O(|V|\log|T| + |T|\log|\Sigma|)$ bits and for the path-compressed trie $O((|V|/\log a)(\log a + \log|T|) + |T|\log|\Sigma|) = O(|V|\log|T| + |T|\log|\Sigma|)$ bits. A query spends $O(|Q|\log|\Sigma|/\log a + \log\log a)$ time in the trie and $O(|Q|\log|\Sigma|/w + \log\log a)$ time in a binary tree. □

### 3.2.4 Summary

Table 3.2 summarizes the properties of the two data structures for ranking strings that we have described. The space requirement is $O(|V|\log|T| + |T|\log|\Sigma|)$ in both cases.

We can simplify the results if we make a couple of reasonable assumptions. First, we assume that the space requirement of the whole task is polynomial in $|T|$. Then, under our model of computation, $w = \Theta(\log|T|)$. Second, we choose the arity of the trie to minimize the query time for query strings of length $q$. With the trie of Theorem 3.11, the choice is such that $\log a = \min\{\log|T|, q\log|\Sigma|\}$. If we optimize for the worst case, i.e., long strings, we get the properties of Table 3.3. Table 3.4 gives the properties when the query string is actually of length $q$.

In the literature, the most common data structure for ranking-like operations on strings is a $|\Sigma|$-ary or binary, possibly path-compressed trie without advanced child access and ranking techniques. The higher arities have been ignored because of the assumption that a character comparison requires $O(1)$ time (i.e., no character packing). The advanced child access and ranking techniques have been ignored because of the assumption $|\Sigma| = O(1)$.

## 3.3 One-dimensional range searching

As was shown in Lemma 2.27, any one-dimensional range query can be reduced to two ranking queries. Any of the data structures of the previous sections can be used. The space and time requirement are the same except an additional $O(k)$ time is needed for reporting $k$ points (assuming points can be handled in constant time).

### 3.3.1 Prefix matching

The prefix matching problem (Problem 2.24) is a special case of range searching in the universe of strings. A query string $Q$ represents the range $[Q, Q'[$, where $Q'$ is the first string following $Q$ in the lexicographic order that does not have $Q$ as a prefix. Computing $Q'$ from $Q$ is not difficult, but it is also not necessary, especially in the case of $|\Sigma|$-ary trie.

Table 3.2: Query times of data structures for ranking strings.

| data structure | query time |
|---|---|
| binary tree for strings (Theorem 3.8) | $\frac{\lvert Q\rvert \log\lvert\Sigma\rvert}{w} + \log\lvert V\rvert$ |
| hybrid path-compressed trie (Theorem 3.11) | $\frac{\lvert Q\rvert \log\lvert\Sigma\rvert}{\log a} + \log\log a$ |

Table 3.3: Query times of the data structures when $w = \Theta(\log|T|)$ and $a$ is chosen to minimize the query time for long (enough) strings.

| data structure | query time |
|---|---|
| binary tree for strings (Theorem 3.8) | $\frac{\lvert Q\rvert \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log\lvert V\rvert$ |
| hybrid path-compressed trie (Theorem 3.11) | $\frac{\lvert Q\rvert \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log\log\lvert T\rvert$ |

Table 3.4: Query times of the data structures when $w = \Theta(\log|T|)$, $|Q| = q$, and $a$ is chosen to minimize the query time for query strings of length $q$.

| data structure | query time |
|---|---|
| binary tree for strings (Theorem 3.8) | $\frac{q \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log\lvert V\rvert$ |
| hybrid path-compressed trie (Theorem 3.11) | $\frac{q \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log(q\log\lvert\Sigma\rvert)$ |

When doing prefix matching in a $|\Sigma_\$|$-ary trie, the child ranking query at the end of a search is not needed at all. Ordinarily, a child ranking query is needed when a node on the search path is found to be missing due to the pruning. However, in the trie $CSTrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$ with the prefix query $Q$, such an incident means that there are no strings with the prefix $Q$, and an empty set can be returned right away. Furthermore, if the node $v$, whose representative prefix is $Q$, does exist, the range of $v$ is exactly $[Q, Q'[$. Thus, the ranks of both the beginning and the end of the range can be stored into the node $v$ (or, if $v$ is implicit, into the next explicit descendant of $v$).

Applying the above idea to the trie of Lemma 3.10 leads to the following result.

**Theorem 3.12 (Prefix matching trie)** *A set $V$ of substrings of $T \in \Sigma^*$ can be stored in $O(|V|\log|T| + |T|\log|\Sigma|)$ bits so that the strings of $V$ with a prefix $Q$ can be reported in $O(|Q| + k)$ time, where $k$ is the number of strings in the answer.*

**Proof** Substitute $|\Sigma|$ for $a$ in Lemma 3.10 and remove the child ranking query time $O(\log\log a)$ from the query time. □

**Remark 3.13** The above optimization requires that, if the range of $Q$ is nonempty, the range has a representative node in the trie. This means that the arity of the trie must be $|\Sigma_\$|$ or an integer root of $|\Sigma_\$|$. In particular, the arity cannot be larger than $|\Sigma_\$|$.

## 3.4 String matching

As was mentioned in Section 2.3.2, a string matching query can be reduced to a prefix matching query. The set $V$ is the set of all suffixes of the text $T$. Thus all the above data structures for prefix matching and string ranking apply to string matching, too. The time and space requirements are the same except an additional $O(k)$ time, for reporting $k$ occurrences.

### 3.4.1 Suffix indexes

The most commonly used text index for string matching is the *suffix tree*, which was first described by Weiner [Wei73]. It is essentially the compact trie $CSTrie_{|\Sigma_\$|}(\Sigma_\$^\infty, V)$, where $V$ is the set of suffixes of the text $T$. A suffix tree usually includes additional pointers called suffix links, which are used in fast construction algorithms [McC76, Ukk95] and some other algorithms [Ukk93, Cob95], but are not needed in basic string matching.

The implementation of the child access queries in suffix trees is often left unspecified on the account that the arity (the alphabet size) is constant. However, if such an assumption cannot be made, hashing is the optimal method. The data structure of Theorem 3.12, when storing the set of all suffixes of the text $T$, can be considered to be a hash implementation of the suffix tree. Using hashing for child access was first suggested by McCreight [McC76].

Besides suffix trees, there are several other string matching indexes based on searching in the set of suffixes of the text. Implementing the binary tree of Theorem 3.8 as a binary search on a sorted array leads to a data structure called the *suffix array*. In fact, Manber and Myers designed their fast binary search for strings precisely for the case of suffixes of a text [MM93]. Colussi and De Col [CD96] have described a hybrid data structure that combines suffix trees and suffix arrays and is quite similar to the data structure of Theorem 3.11.

Other suffix-based structures include DAWG, CDAWG, suffix cactus, and LC-trie. The DAWG (directed acyclic word graph, also known as suf-

Table 3.5: Data structures for string matching.

| data structure | query time | space |
|---|---|---|
| suffix array (Theorem 3.8) | $\frac{\lvert P\rvert \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log\lvert T\rvert + k$ | $\lvert T\rvert \log\lvert T\rvert$ |
| hybrid suffix tree (Theorem 3.11) | $\frac{\lvert P\rvert \log\lvert\Sigma\rvert}{\log\lvert T\rvert} + \log\log\lvert T\rvert + k$ | $\lvert T\rvert \log\lvert T\rvert$ |
| suffix tree (Theorem 3.12) | $\lvert P\rvert + k$ | $\lvert T\rvert \log\lvert T\rvert$ |

fix automaton) is an automaton recognizing all substrings (or suffixes) of the text [BBH+85, Cro86]. It can also be described as the string trie compressed by combining identical subtrees. The CDAWG (Compact DAWG) is a DAWG with path compression [BBH+87, CV97]. The suffix cactus is a variation of the suffix tree where the path compression is taken a step further [Kär95]. The LC-trie (level-compressed trie) [AN93] is a binary trie where several levels may be compressed in to one level. This creates the effect of a trie with varying arity. In [AN95] level compression is combined with path compression to implement suffix trees. None of these data structures offers asymptotic advantages compared to the solutions we have described.

The properties of the solutions to string matching are summarized in Table 3.5. We have made the same simplifying assumptions as in Table 3.3, i.e., $w = \Theta(\log|T|)$, and the query time of the tries is minimized for long query strings. Since $|V| = |T|$ and $\log|\Sigma| = O(w) = O(\log|T|)$, the space requirement simplifies to $O(|T|\log|T|)$.

### 3.4.2 Sparse suffix trees

The space requirement for all the above solutions is $\Omega(|T|\log|T|)$ bits. This is not optimal since the text itself needs only $O(|T|\log|\Sigma|)$ bits, which can be significantly less if the alphabet is small.

A simple way to reduce the space requirement is to store nothing but the text. This leads to the algorithms that scan the whole text, for example, the Knuth–Morris–Pratt algorithm [KMP77]. Then, the space requirement is the optimal $O(|T|\log|\Sigma|)$ bits but the query time increases to $\Omega(|T|\log|\Sigma|/\log|T|)$.

The $\Omega(|T|\log|T|)$ space requirement in the previous structures is due to the need to have $\Omega(|T|)$ pointers to the text to represent the suffixes. This is unavoidable for any data structure based on searching the set of all suffixes. One attempt to get around this problem is the sparse suffix tree [KU96b], which stores only a subset of the suffixes.

Table 3.6: Space–time tradeoff of the sparse suffix tree.

| space | query time | $h$ |
|---|---|---|
| $\lvert T\rvert \frac{\log \lvert T\rvert \log \lvert\Sigma\rvert}{\log\log\log \lvert T\rvert}$ | $\lvert P\rvert \log\log \lvert T\rvert + \mathrm{k}$ | $\log_{\lvert\Sigma\rvert} \log\log \lvert T\rvert + 1$ |
| $\frac{1}{\varepsilon}\lvert T\rvert \frac{\log \lvert T\rvert \log \lvert\Sigma\rvert}{\log\log \lvert T\rvert}$ | $\lvert P\rvert \log^{\varepsilon} \lvert T\rvert + k$ | $\varepsilon \log_{\lvert\Sigma\rvert} \log \lvert T\rvert + 1$ |
| $\frac{1}{\varepsilon}\lvert T\rvert \log \lvert\Sigma\rvert$ | $\lvert P\rvert \lvert T\rvert^{\varepsilon} + k$ | $\varepsilon \log_{\lvert\Sigma\rvert} \lvert T\rvert + 1$ |

Let $V_h$ be the set of every $h$th suffix of a text $T$, i.e., $V = \{T[1, |T|], T[h+1, |T|], T[2h+1, |T|], \ldots\}$. A path compressed trie for $V_h$ is called a *$h$-spaced sparse suffix tree*. Its space requirement is $O(|T|((\log |T|)/h + \log |\Sigma|))$ bits. With it we can easily find the occurrences of a string pattern $P$ starting at positions $1, h+1, 2h+1, \ldots$. We call these the $h$-occurrences of $P$. The remaining problem is to find the other occurrences.

In [KU96b], two basic methods are given for finding the other occurrences. The first method finds all $h$-occurrences of the pattern suffixes $P[i, |P|]$, $i \in [2, h]$, and then checks which of those occurrences are a part of a full pattern occurrence. This method has two problems. First, it does not work if $|P| < h-1$. Second, the number of $h$-occurrences of the pattern suffixes could be much larger than the number of occurrences of $P$. Thus, the worst case behavior is very bad. In [KU96b], the focus is on the average case behavior.

The second method generates a set of patterns by prepending $P$ with each of the strings in $\cup_{i=1}^{h-1} \Sigma^i$, and finds all $h$-occurrences of those patterns. Each occurrence found contains a distinct occurrence of $P$ as a suffix. If the $h$-spaced sparse suffix tree has arity $|\Sigma|$, we can, instead of generating all the patterns, start matching $P$ from every explicit or implicit node of the tree up to depth $h-1$. We can also use the technique of Theorem 3.12 to avoid the final child ranking queries. This leads to the total query time of $O(|P||\Sigma|^{h-1} + k)$. We have the following result.

**Theorem 3.14 (Sparse suffix tree)** *A text $T \in \Sigma^*$ can be stored in $O(|T|((\log |T|)/h + \log |\Sigma|))$ bits so that all occurrences of a string pattern $P$ can be found in $O(|P||\Sigma|^{h-1} + k)$ time, where $h \geq 1$ and $k$ is the number of occurrences.*

With different choices for $h$ we can have different space–time tradeoffs. Table 3.6 gives some examples.

## 3.5 Semi-infinite two-dimensional range searching

Next, we consider the problem of semi-infinite two-dimensional range searching, the easier special case of the general two-dimensional range searching.

In most of what follows, we assume that the transformation into rank space has been made. To get a general result, the time and space of ranking the coordinates needs to be added to the time and space requirements. What those are depends on the universe and method used for ranking. For some results below, we also give a general version for an integer universe, i.e., a universe whose points can be stored in constant number of words.

In the rank space case, $V$ is a set of points in the two-dimensional universe $[1,|V|]^2$. Furthermore, let $V_1$ be the set of first coordinates and $V_2$ the set of second coordinates of $V$. Then we have $V_1 = V_2 = [1,|V|]$. We consider only queries, where $h_2 = |V|$, i.e., the query range is $R = R_1 \times R_2 = [l_1, h_1] \times [l_2, |V|]$. Queries with a different side open can be treated symmetrically.

### 3.5.1 Priority search tree

We will first present a data structure called the *priority search tree* [McC85]. Consider the binary trie $Trie_2([1,|V|], V_1)$, i.e., the tree $Tree_2([1,|V|])$ with the leaves representing $V_1$. As was discussed in Section 3.1.2, each node of the trie represents a range of $[1,|V|]$. Let the *border* of the range $R_1 = [l_1, h_1]$, denoted by $B(R_1)$, be the set of nodes whose range contains one or both of $l_1$ and $h_1$. Let the *cover* of $R_1$, denoted by $C(R_1)$, be the set of nodes whose range is nested in $R_1$ (i.e., contained by $R_1$ but not containing $l_1$ or $h_1$) but whose parent's range is not nested in $R_1$ (i.e., the parent contains $l_1$ or $h_1$). The border and cover are illustrated in Figure 3.3.

The ranges of the cover nodes do not overlap and their union is $R_1 \setminus \{l_1, h_1\}$. All proper ancestors of the nodes in $B(R_1)$ and $C(R_1)$ are in $B(R_1)$. On any level of the trie, there are at most two border nodes and at most two cover nodes. Thus, $|B(R_1)|, |C(R_1)| = O(\log|V|)$ and they can be computed in $O(\log|V|)$ time by a simple search starting from the root.

We will store the set $V$ into $Trie_2([1,|V|], V_1)$ as follows. Into each node $u$, we store the point with the largest second coordinate such that the first coordinate is in the range of $u$, and the point is not stored into any ancestor of $u$. Some of the nodes will be empty. The resulting data structure, *priority search tree*, is denoted $PTree_2(V)$.

A semi-infinite range query in $PTree_2(V)$ is executed in two phases:

1. find the points that are stored in the nodes of $B(R_1)$ (the points "on the border") and are in the range $R$, and

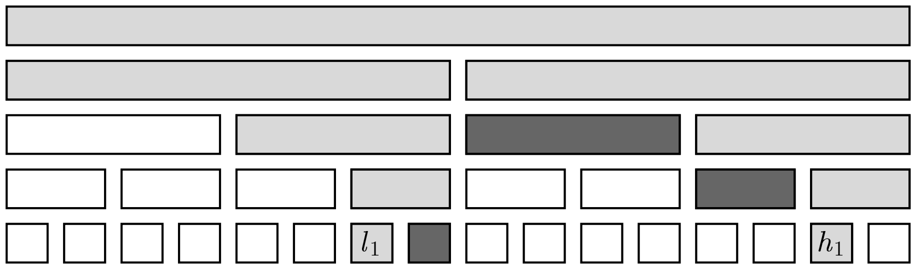



Figure 3.3: The border (light gray) and the cover (dark gray) of the range $[l_1, h_1]$ in a trie. The horizontal dimensions of a node represent its range.

2. find the points that are stored in the subtrees rooted at the nodes of $C(R_1)$ (the points "under the cover") and have a second coordinate at least $l_2$.

The points under the cover have the first coordinate in the range $R_1$; thus checking just the second coordinate is enough. The points not under the cover and not on the border have a first coordinate outside $R_1$ and can be ignored.

The first phase is executed simply by comparing each point on the border to the query range $R$. During the second phase, let *active nodes* be the roots of the subtrees to be searched. Let $u$ be an active node, and let $v = (v_1, v_2)$ be the point stored in $u$. If $v_2 \geq l_2$, $v$ is reported as an answer, and the children of $u$ are activated. If $v_2 < l_2$, all points in the subtree under $u$ have second coordinate less than $l_2$ and the search of that subtree can be stopped.

In the beginning of the second phase, there are $O(\log |V|)$ active nodes (the cover nodes). A successful comparison ($v_2 \geq l_2$) may increase the number of active nodes by one but also reports a point. A failed comparison ($v_2 < l_2$) reduces the number of active nodes by one. Thus, during the second phase, there is $O(k)$ successful comparisons and $O(\log |V| + k)$ failed comparisons, where $k$ is the number of points reported.

We have the following result.

**Lemma 3.15 (Priority search tree)** *A set $V$ in the two-dimensional rank space can be stored in $O(|V| \log |V|)$ bits such that a semi-infinite two-dimensional range query can be answered in $O(\log |V| + k)$ time, where $k$ is the size of the answer.*

By doing a rank transformation before the range search using, e.g., binary trees, we have the following result.

**Corollary 3.16** *A set $V$ in a two-dimensional integer universe $U$ can be stored in $O(|V|\log|U|)$ bits so that a semi-infinite two-dimensional range query can be answered in $O(\log|V|+k)$ time, where $k$ is the size of the answer.*

### 3.5.2 Improved priority search tree

There is also another way to characterize the sets $B(R_1)$ and $C(R_1)$ in $Trie_2([1,|V|],V_1)$. Let $u_l$ and $u_h$ be the leafs representing $l_1$ and $h_1$, respectively, and let $u'$ be their nearest common ancestor. Note that, due to the regular structure of $Trie_2([1,|V|],V_1)$, $u_l$ and $u_h$ can be computed in constant time, and $u'$ in $O(\log\log|V|)$ time by a binary search on the paths leading to $u_l$ and $u_h$ (or in constant time using the methods in [HT84, SV88]). Let $B(u)$ be the set of ancestors of a leaf $u$. Then, $B(R_1)=B(u_l)\cup B(u_h)$. Let $C_{[}(u_l,u')$ be the set of nodes that are right children of the nodes on the path from $u'$ to $u_l$ but are not themselves on the path. Symmetrically, let $C_{]}(u_h,u')$ be the set of nodes that are left children of the nodes on the path from $u'$ to $u_h$ but are not themselves on the path. Then $C(R_1)=C_{[}(u_l,u')\cup C_{]}(u_h,u')$.

In [Ove88], Overmars describes how to speed up queries in a priority search tree by augmenting it with additional structures. Overmars' algorithm does not compute the border and the cover explicitly, but it does execute the same two phases of the search.

To do the first phase, we use the fact that $B(R_1)=B(u_l)\cup B(u_h)$. We store the points in $B(u_l)$ and $B(u_h)$ into separate priority search trees. The first phase is done by querying the two priority search trees. We need to have $O(|V|)$ priority search trees (one for each leaf), each storing $O(\log|V|)$ points. By Corollary 3.16, the query time is $O(\log\log|V|+k)$ and the total space requirement $O(|V|\log^2|V|)$.

The second phase is done as before except that, in the beginning, we do not activate all cover nodes but only the *successful cover nodes*, the cover nodes where the comparison will be successful. There are $O(k)$ successful cover nodes, and thus only $O(k)$ nodes are ever activated. To find the successful cover nodes, we use the fact that $C(R_1)=C_{[}(u_l,u')\cup C_{]}(u_h,u')$. The sets $C_{[}(u_l,u')$ and $C_{]}(u_h,u')$ are stored in lists in decreasing order of the second coordinate of the points in the nodes. Then the successful cover nodes are in the beginning of the lists and can be found in $O(k)$ time.

For each pair of a leaf $u$ and its ancestor $u'$, we need the lists for $C_{[}(u,u')$

and $C_{]}(u, u')$. There are $O(|V| \log |V|)$ such pairs, and each list contains $O(\log |V|)$ nodes. Thus, the space requirement of the lists is $O(|V| \log^3 |V|)$ bits.

As the final step, the space requirement is reduced from $O(|V| \log^3 |V|)$ to $O(|V| \log |V|)$ by pruning. Let $\ell$ be a level such that the subtrees with the root at level $\ell$ have $\Theta(\log^2 |V|)$ nodes. Collapse all the subtrees with the root at level $\ell$ into a single leaf. The points in the collapsed subtree are stored in the new leaf. The search for the points stored in the remaining internal nodes is done as before. Since there are now only $O(|V|/\log^2 |V|)$ leafs, all the data structures fit into $O(|V| \log |V|)$ bits.

We still need to search the points in the leaves. The $O(\log^2 |V|)$ points in a leaf $u$ are stored in two data structures, a priority search tree (Corollary 3.16) and a list in decreasing order of the second coordinates. With $O(|V|/\log^2 |V|)$ leaves with $O(\log^2 |V|)$ points in each, the total space requirement is $O(|V| \log |V|)$ bits.

For the first phase of a query, the interesting leaves are the one or two leaves on the border, i.e., the leaves whose ranges contains $l_1$ or $h_1$. The priority search trees in those leaves are searched in $O(\log \log |V| + k)$ time. For the second phase, the interesting leaves are the ones that get activated during the search. For those leaves, we search the list; the points in the range are in the beginning of the list. This requires $O(k)$ time. We have the following result due to Overmars [Ove88].

**Theorem 3.17 (Improved priority search tree)** *A set $V$ in the two-dimensional rank space can be stored in $O(|V| \log |V|)$ bits so that a semi-infinite two-dimensional range query can be answered in $O(\log \log |V| + k)$ time, where $k$ is the size of the answer.*

**Remark 3.18** Overmars [Ove88] also gives a data structure for doubly semi-infinite range queries. The data structure is simpler but has the same asymptotic space and time requirement.

## 3.6 General two-dimensional range searching

Finally, we describe solutions to the problem of general two-dimensional range searching in rank space. The solutions can be extended to arbitrary two-dimensional universes by providing a method for transforming coordinates into the rank space.

In the rank space case, $V$ is a set of points in the two-dimensional universe $[1, |V|]^2$. As before, $V_1$ is the set of first coordinates of $V$, $V_2$ is the set of second coordinates of $V$, and $V_1 = V_2 = [1, |V|]$. The query range is $R = R_1 \times R_2 = [l_1, h_1] \times [l_2, h_2]$.

### 3.6.1 A method by Chazelle

Consider again the trie $Trie_2([1, |V|], V_1)$. For each node $u$ of the trie, let $V_u$ be the points of $V$ with the first coordinate in the range of $u$, ordered by the second coordinate. For a coordinate value $i \in V_2$ and a node $u$, let $r(i, u)$ be the rank of $i$ in the second coordinates of $V_u$. A range query can now be answered as follows:

1. Compute the cover $C(R_1)$.
2. For each cover node $u$, compute $r(l_2, u)$ and $r(h_2, u)$.
3. For each cover node $u$, report the points of $V_u$ with the ranks in the range $[r(l_2, u), r(h_2, u)]$.

In addition, we need to check the two points in the leafs representing $l_1$ and $h_1$.

A straightforward implementation of the above procedure is to store, into every node $u$, the set $V_u$ in an array ordered by the second coordinate. Then step 1 can be done in $O(\log |V|)$ time as described in the previous section, step 2 in $O(\log^2 |V|)$ time by binary searching, and step 3 in $O(k)$ time. Since each point is stored once on every level, the space requirement is $O(|V| \log^2 |V|)$ bits. This is the *range tree* of Bentley [Ben80].

In [Cha88], Chazelle describes a different implementation for the above procedure outline. The first step remains the same, but the second and third steps are different. Chazelle's solution for the second step is based on computing each rank $r(i, u)$ from $r(i, u')$, where $u'$ is the parent of $u$. This is, in fact, a ranking operation, where $V_{u'}$ is the universe and $V_u$ the subset. Thus this operation can be done in $O(1)$ time and in $O(|V_{u'}|)$ bits of space by using the bucketed bitmap (Theorem 3.1). The space requirement of the bucketed bitmaps is $O(|V|)$ bits for every level of the trie, a total of $O(|V| \log |V|)$ bits. The step 2 is executed by first computing $r(l_2, u')$ and $r(h_2, u')$ for the root $u'$ and then propagating the ranks downwards to the cover nodes in $O(\log |V|)$ steps. This improves the total query time to $O(\log |V| + k)$. The space requirement for steps 1 and 2 is $O(|V| \log |V|)$ bits, but step 3 still needs $O(|V| \log^2 |V|)$ bits.

The space requirement of step 3 can also be reduced. Let $u$ be a cover node and let $i$ be an integer such that $r(l_2, u) \leq i \leq r(h_2, u)$. Then the $i$th point $v$ in $V_u$ is in the range $R$. After step 2, we know the rank $i$ of the point but not the actual point. We do not want to spend $O(|V_u| \log |V|)$ bits for a lookup table to find out the point. Instead we have a table of $O(|V_u|)$ bits, where the $i$th bit tells which child of $u$ contains the point $v$. Then we can use the technique from step 2 to compute the rank of $v$ in that child. The rank is propagated downwards until a leaf is reached, and only there the actual point is stored. Thus, space requirement is reduced to $O(|V|\log|V|)$ but the time requirement now increases to $O((k+1)\log|V|)$.

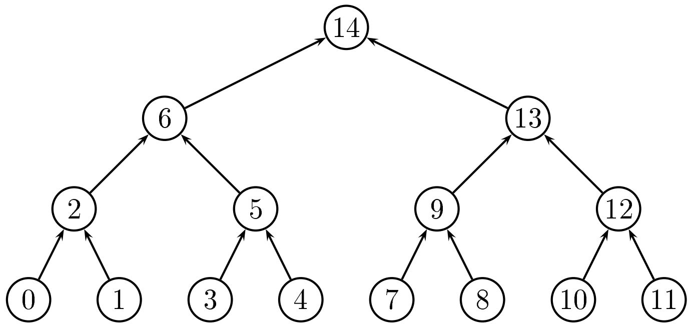
In [Cha88], Chazelle also describes data structures for step 3 that reduce the time requirement and allow a space–time tradeoff. The idea is to propagate the ranks downwards in the tree several levels at a time. To jump $h$ levels down, we have to find out, in which of the $2^h$ descendants of $u$ the point $v$ is, and what is $v$'s rank in that descendant. This can be done with the data structure of Lemma 3.2 in $O(1)$ time and $O(|V_u|h)$ bits of space.

As the final matter in Chazelle's data structure, we need to decide what jumps to implement. Number the levels of the trie from 0 to $\lceil\log|V|\rceil-1$ with 0 being the root level. Organize the level numbers in an $a$-ary tree, called the *level tree*. In the level tree, each subtree represents a consecutive range of level numbers. The largest number in each subtree is at the root. Then, the root of the level tree contains $\lceil\log|V|\rceil-1$, the number of the leaf level of the trie (the level we want to end up at). We call the levels of the level tree *tiers* to avoid confusion. There are $O(\log_a\log|V|)$ tiers in total. A tier $i$ has $a^i$ level numbers about evenly distributed over $[0,\lceil\log|V|\rceil-1]$. Figure 3.4 shows examples of level trees.

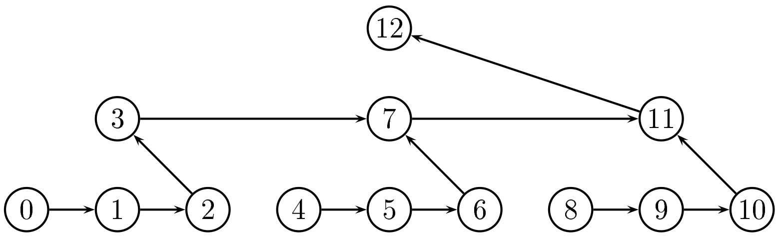
Let a level number $\ell+h$ be the parent of a level number $\ell$ in the level tree. We implement a jump from every node on the level $\ell$ to the level $\ell+h$. This is illustrated in Figure 3.4(a). Now any point can be propagated to a leaf in $O(\log_a\log|V|)$ time. There are $a^i$ jumps from the tier $i$ to the tier $i-1$ with an average length of $O(\log|V|/a^{i-1})$. Thus the space requirement of the jumps for one tier is $O(a|V|\log|V|)$ and the total space requirement is then $O(a|V|\log|V|\log_a\log|V|)$ bits.

**Theorem 3.19** *For any $a\in[2,\log|V|]$, a set $V$ in the two-dimensional rank space can be stored in $O(a|V|\log|V|\log_a\log|V|)$ bits so that a two-dimensional range query can be answered in $O(\log|V|+k\log_a\log|V|)$ time, where $k$ is the size of the answer.*

Another way to organize the jumps is based on the same level tree, but now it is implemented using sibling lists. That is, there is a jump to a parent only from the child that is closest to the parent. Other nodes have a jump to the nearest sibling towards the parent. This is illustrated in Figure 3.4(b). Now a point propagation can be done in $O(a\log_a\log|V|)$ time. All jumps from a tier $i$ level either to another tier $i$ level or to a




(a) Binary level tree with jumps of Theorem 3.19




(b) Ternary level tree with jumps of Theorem 3.20

Figure 3.4: Level trees. The arrows represent the jumps.

tier $i-1$ level has length $O(\log|V|/a^i)$. Thus the space requirement of the jumps for one tier is $O(|V|\log|V|)$ bits and the total space requirement is then $O(|V|\log|V|\log_a\log|V|)$ bits.

**Theorem 3.20** *For any $a \in [2, \log|V|]$, a set $V$ in the two-dimensional rank space can be stored in $O(|V|\log|V|\log_a\log|V|)$ bits so that a two-dimensional range query can be answered in $O(\log|V|+ka\log_a\log|V|)$ time, where $k$ is the size of the answer.*

### 3.6.2 A method by Overmars

The next data structure we describe was introduced by Overmars [Ove88]. Let $\ell$ be the first level of the trie $Trie_2([1,|V|],V_1)$ such that $l_1$ and $h_1$ are in the ranges of two different nodes $u_l$ and $u_h$, respectively. In other words, $u_l$ and $u_h$ have a common parent $u'$ whose range contains both $l_1$ and $h_1$. The sets $V_{u_l}$ and $V_{u_h}$ contain all the points with the first coordinate in the range

$R_1$ but may also contain other points. However, all points in $V_{u_l}$ have the first coordinate smaller than $h_1$ and all points in $V_{u_h}$ have the first coordinate larger than $l_1$. Thus the range query $[l_1, h_1] \times [l_2, h_2] \cap V$ is reduced to the semi-infinite range queries $[l_1, |V|] \times [l_2, h_2] \cap V_{u_l}$ and $[1, h_1] \times [l_2, h_2] \cap V_{u_h}$. For every node $u$, we have $y$-fast tries (Theorem 3.7) for transforming the semi-infinite ranges into the reduced rank space of $V_u$, and improved priority search trees (Theorem 3.17) for executing the semi-infinite range query. We have the following result due to Overmars [Ove88].

**Lemma 3.21** *A set $V$ in the two-dimensional rank space can be stored in $O(|V| \log^2 |V|)$ bits so that a two-dimensional range query can be answered in $O(\log\log |V| + k)$ time, where $k$ is the size of the answer.*

Finally, we describe a new variation of the above data structure that reduces the space requirement at the expense of the query time. The data structure is the same as above except the trie $Trie_2([1, |V|], V_1)$ is replaced with the $a$-ary $Trie_a([1, |V|], V_1)$. This reduces the number of levels to $O(\log_a |V|)$ and thus the space to $O(|V| \log |V| \log_a |V|)$ bits. Now the nodes $u_l$ and $u_h$ may not cover the whole range $R_1$; there can be up to $a-2$ nodes, denoted by $C'(R_1)$, between $u_l$ and $u_h$ on the same level and with the range completely inside $R_1$.

To execute a range query, we do semi-infinite range queries in $u_l$ and $u_h$ as before. In addition, for every $u \in C'(R_1)$, we find the points of $V_u$ with the second coordinate in the range $[l_2, h_2]$. Thus the remaining problem is to compute the ranks $r(l_2, u)$ and $r(h_2, u)$ for each $u \in C'(R_1)$. This is done as follows.

Let $u'$ be the common parent of the nodes in $C'(R_1)$. First, the ranks $r(l_2, u')$ and $r(h_2, u')$ are computed with a $y$-fast trie in time $O(\log\log |V|)$. Then the ranks are propagated into the children as follows. Divide $V_{u'}$ into buckets of size $a$. For each bucket boundary point $p$ and each child $u$ of $u'$, store $r(p, u)$ into a table of $|V_{u'}|/a \times a = |V_{u'}|$ entries and with the space requirement of $O(|V_{u'}| \log |V|)$ bits. For each point $v \in V_{u'}$, store the child $u$ of $u'$ whose range contains $v$; this fits in $O(|V_{u'}| \log |V|)$ bits, too. To compute the ranks of a point $v$ in each child of $u'$, find the bucket boundary nearest to $r(v, u')$, and update the bucket boundary ranks by scanning, one by one, the points between $r(v, u')$ and the bucket boundary. This requires $O(a)$ time. Thus the ranks $r(l_2, u)$ and $r(h_2, u)$ for every child $u$ of $u'$ can be computed in $O(\log\log |V| + a)$ time and $O(|V_{u'}| \log |V|)$ bits of space. Those children include the set $C'(R_1)$.

Every point of $V$ is stored once on every level of $Trie_a([1, |V|], V_1)$, and the data structures require no more space than the points themselves.

Table 3.7: Space–time tradeoff for two-dimensional range searching.

| space | query time | theorem | $a$ |
|---|---|---|---|
| $\frac{\|V\| \log^2 \|V\|}{\log\log\log \|V\|}$ | $\log\log \|V\| + k$ | 3.22 | $\log\log \|V\|$ |
| $\frac{1}{\varepsilon} \frac{\|V\| \log^2 \|V\|}{\log\log \|V\|}$ | $\log^{\varepsilon} \|V\| + k$ | 3.22 | $\log^{\varepsilon} \|V\|$ |
| $\frac{1}{\varepsilon}\|V\| \log^{1+\varepsilon} \|V\|$ | $\log \|V\| + \frac{1}{\varepsilon}k$ | 3.19 | $\log^{\varepsilon} \|V\|$ |
| $\|V\| \log \|V\| \log\log \|V\|$ | $\log \|V\| + k \log\log \|V\|$ | 3.19 or 3.20 | 2 |
| $\frac{1}{\varepsilon}\|V\| \log \|V\|$ | $\log \|V\| + \frac{1}{\varepsilon} k \log^{\varepsilon} \|V\|$ | 3.20 | $\log^{\varepsilon} \|V\|$ |
| $\frac{1}{\varepsilon}\|V\| \log \|V\|$ | $\|V\|^{\varepsilon} + k$ | 3.22 | $\|V\|^{\varepsilon}$ |

Thus the space requirement is $O(|V| \log |V| \log_a |V|)$. We have the following result.

**Theorem 3.22** *For any $a \in [2, |V|]$, a set $V$ in the two-dimensional rank space can be stored in $O(|V| \log |V| \log_a |V|)$ bits so that a two-dimensional range query can be answered in $O(a + \log\log |V| + k)$ time, where $k$ is the size of the answer.*

### 3.6.3 Summary

Theorems 3.19, 3.20 and 3.22, provide us with many possible solutions to the two-dimensional range search problem with different space–time tradeoffs. Table 3.7 gives some of these solutions ordered by the space requirement.

# Chapter 4

# Repetition-based indexing

Repetition-based indexing is a scheme for solving the indexed pattern matching problem (Problem 2.22). A key feature of repetition-based indexing is its flexibility with respect to the tradeoff between the size of the index structure and the time for answering pattern queries. The possibilities of the scheme range from the bare text with no index to suffix-tree-like data structures. More importantly, it also provides alternatives between these two extremes. The only other solution presented so far between these extremes is the sparse suffix tree (Section 3.4.2).

In this chapter, we describe the basic ideas behind repetition-based indexing and discuss the extremes of the space–time tradeoff. A compromise between the extremes, called the Lempel–Ziv index, will be presented in detail in Chapter 5. A variation of the Lempel–Ziv index for the $q$-gram matching problem is presented in Chapter 6. The idea of repetition-based indexing first appeared in [KU96a] as the special case described in Chapter 5.

## 4.1 The general framework

We start by describing the idea of repetition-based indexing at a general level.

### 4.1.1 Phrases

Repetition-based indexing takes advantage of repetitions in the text. By a repetition we mean a string that appears two or more times in the text. The information about repetitions is represented by phrases.

**Definition 4.1** Let $T$ be a string and let $I$ and $J$ be intervals of $T$ such that $T(I) = T(J)$. Then $I \to J$ is a *phrase* of $T$. The interval $I$ is called

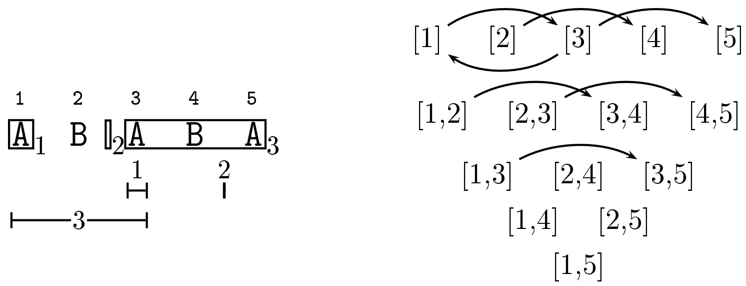



Figure 4.1: On the left is an illustration of the phrase set $Z = \{[3] \to [1], [5,4] \to [3,2], [1,3] \to [3,5]\}$ for the text $T =$ ABABA. The sources are shown as line segments under the text and the repeats as boxes around text segments. On the right is the phrase graph $G_Z$.

the *source* and $J$ the *repeat*. The phrase is *trivial* if $I = J$.

The two intervals of a phrase are in symmetric position, that is, if $I \to J$ is a phrase, then $J \to I$ is a phrase, too. The *direction* of the phrase, i.e., which interval is the source and which the repeat, affects the way phrases are used. In algorithms, a phrase is accessed through the source. That is, algorithms search for phrases whose source contains the pattern, and then use the information that the repeat has the same content.

A long phrase implicitly defines many shorter phrases as follows.

**Definition 4.2** Let $T$ be a string and $I \to J$ its phrase. Let $I'$ and $J'$ be intervals of $T$ such that $|I'| = |J'|$, $I' \subseteq I$, $J' \subseteq J$, and $\overleftarrow{I'} - \overleftarrow{I} = \overleftarrow{J'} - \overleftarrow{J}$. Then $I' \to J'$ is a *subphrase* of $I \to J$. A subphrase is always a phrase.

Repetition-based indexing uses the information in a set of phrases. A lot of this information is implicit in the set. It can be made more explicit using phrase graphs.

**Definition 4.3** Let $T$ be a string and let $Z$ be a set of phrases of $T$. The *phrase graph* $G_Z$ for $Z$ is defined as follows. The vertices of $G_Z$ are the nonempty intervals of $T$. There is a directed edge from interval $I$ to interval $J$ if $I \to J$ is a subphrase of some phrase of $Z$.

An example of a phrase set and its phrase graph is shown in Figure 4.1.

The implicit information in a set of phrases is more than just the subphrase information as shown by the following lemma.

**Lemma 4.4** *Let $T$ be a string, $Z$ a set of phrases of $T$, and $G_Z$ the phrase graph of $Z$. Let intervals $I$ and $J$ belong to the same connected component of $G_Z$, i.e., there is an undirected path from $I$ to $J$. Then $T(I) = T(J)$.*

It should be emphasized that phrase graphs are not explicitly represented or stored in repetition-based indexing. They are just a convenient way to think and talk about the implicit information in a set of phrases. For example, when we say that an algorithm follows an edge from a vertex $I$ to a vertex $J$ in a phrase graph $G_Z$, we mean that the algorithm finds a phrase of $Z$ whose source contains $I$ and then computes $J$ as the interval at the corresponding position in the repeat.

The information content of a phrase graph does not depend on the direction of edges. Algorithms, however, travel in the phrase graph only in the direction of edges. This limits significantly the reachability of vertices.

**Definition 4.5** The vertices of a phrase graph with no incoming edges are called *roots*. A phrase graph is *rooted* if every vertex is reachable from a root. A phrase set is rooted if its phrase graph is rooted.

A phrase graph that is not rooted must be cyclic, but a cyclic phrase graph can be rooted, too.

In a rooted phrase graph one has to be concerned with the reachability of roots only; other vertices can be reached by following the edges. The next example illustrates a problem with phrase sets and graphs that are not rooted.

**Example 4.6** Figure 4.2 shows two phrase sets and their phrase graphs. The characters within repeats are hidden, but in a rooted phrase set the hidden characters can be inferred.

In the first case, the phrases are $[7, 8] \to [3, 4]$ and $[2, 4] \to [7, 9]$. The phrase set is rooted and implies that $T[9] = T[4] = T[8] = T[3] = T[7] = T[2] = \texttt{A}$.

In the second case, the phrases are $[8, 9] \to [3, 4]$ and $[2, 4] \to [7, 9]$. The phrase set is not rooted. For instance, $[3]$ is not reachable from a root. The phrase set implies that $T[7] = T[2] = \texttt{A}$. It also implies that $T[3] = T[8]$ and $T[4] = T[9]$ but it is not possible to infer the actual characters at the positions 3, 4, 8, and 9.

### 4.1.2 Repetition-based indexing

Now we are ready to describe the ideas of repetition-based indexing. Given any rooted phrase set of the text, the first phase of a query is to locate the

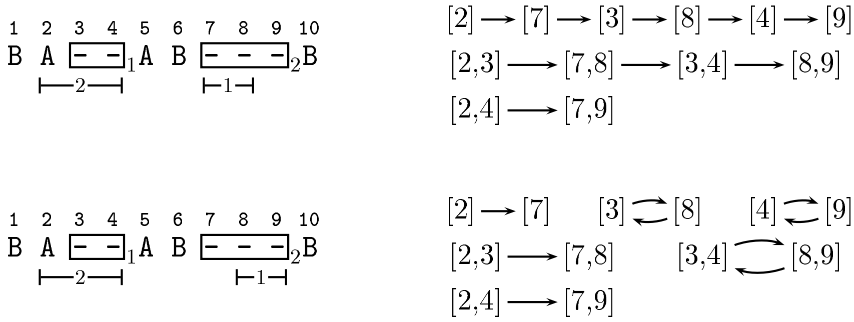



Figure 4.2: Two examples of a string with two phrases (left) and their phrase graphs (right). The unconnected vertices of the phrase graphs are not shown.

occurrences of the pattern that are roots of the phrase graph. Then follow the edges of the phrase graph to find the rest of the occurrences. This is formalized below.

**Definition 4.7** Let $T$ be the text and $Z$ a rooted phrase set of $T$. Let $\mathcal{O}$ be the set of all occurrences of a pattern $P$ in $T$. The set $\mathcal{O}'$ of *primary occurrences* of $P$ consists of the occurrences that are not contained by any repeat of $Z$ (i.e., the occurrences that are roots of $G_Z$). The set $\mathcal{O}''$ of *secondary occurrences* of $P$ is the complementary set $\mathcal{O} \setminus \mathcal{O}'$, i.e., the occurrences that are contained by some repeat of $Z$. Formally,

$$\mathcal{O}' = \{O \in \mathcal{O} \mid \forall I \to J \in Z : O \nsubseteq J\}$$
$$\mathcal{O}'' = \mathcal{O} \setminus \mathcal{O}' = \{O \in \mathcal{O} \mid \exists I \to J \in Z : O \subseteq J\}.$$

**Example 4.8** Consider the text and the phrase set in Figure 4.3. The pattern ABB has five occurrences. The occurrences at $[2, 4]$ and $[21, 23]$ are primary and the occurrences at $[8, 10]$, $[15, 17]$, and $[25, 27]$ are secondary.

**Problem 4.9 (Primary search)** Let $T$ be the text and $Z$ a rooted phrase set of $T$. Given a pattern $P$, the *primary search query* asks for all primary occurrences of $P$ in $T$. Let $\mathcal{P}$ be a class of patterns. The *primary search problem* is to store $T$ and $Z$ in such a way that primary search queries can be answered quickly for any pattern in $\mathcal{P}$. The data structure supporting primary search queries is called the *primary index*.

**Problem 4.10 (Secondary search)** Let $T$ be the text and $Z$ a rooted phrase set of $T$. Given the set of the primary occurrences of a pattern $P$

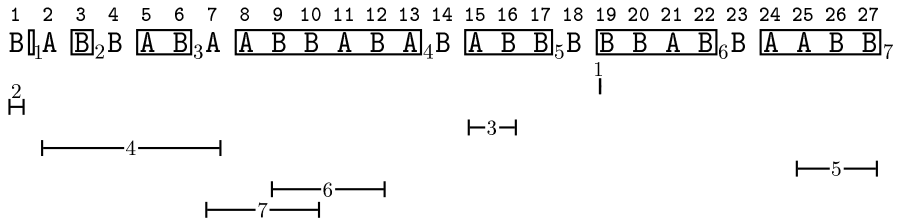


Figure 4.3: A phrase set for the string `BABBABAABBABABABBBBBABBAABB`.

in $T$, the *secondary search query* asks for all secondary occurrences of $P$ in $T$. Let $\mathcal{P}$ be a class of patterns. The *secondary search problem* is to store $Z$ in such a way that secondary search queries can be answered quickly for any pattern in $\mathcal{P}$. The data structure supporting secondary search queries is called the *secondary index*.

## 4.2 Realizations

To make repetition-based indexing concrete, we need to define the phrase set, and give algorithms for primary and secondary searching. Let us first consider the choice of the phrase set.

### 4.2.1 The phrase set

The two main goals in choosing the phrase set for a text are to minimize the number of phrases and to minimize the number of primary occurrences. The motivation for the first goal is to minimize the space requirement of the primary and secondary indexes. Both indexes generally have (at least) linear dependence on the number of phrases. The reason for the second goal is that primary searching, in general, is harder than secondary searching. Therefore, the phrase repeats should cover the intervals of the text as well as possible to reduce the primary searching time.

The two goals are contradictory, but some things can be done to advance one goal without hurting the other. First, phrases that do not contribute to the coverage can be removed. Below is the definition of such phrases.

**Definition 4.11** A phrase $I \to J$ of a rooted phrase set $Z$ is *redundant* if removing it from $Z$ keeps the phrase graph $G_Z$ rooted and the set of roots unchanged.

**Lemma 4.12** *A nonempty phrase is redundant if and only if its repeat is contained by another repeat.*

Second, the phrases should be as long as possible. Some notions of (local) maximality of phrases are defined in the following.

**Definition 4.13** Let $T$ be a string and $I \to J$ its phrase. The phrase $I \to J$ is *right maximal* if $[\overleftarrow{I}, \overrightarrow{I}+1] \to [\overleftarrow{J}, \overrightarrow{J}+1]$ is not a phrase of $T$. The phrase $I \to J$ is *left maximal* if $[\overleftarrow{I}-1, \overrightarrow{I}] \to [\overleftarrow{J}-1, \overrightarrow{J}]$ is not a phrase of $T$. The phrase is *maximal* if it is both left maximal and right maximal.

Third, the overlap of the repeats should be minimal. Overlap wastes coverage because some intervals are covered multiple times. Multiple coverage also makes secondary search more difficult.

One of the two goals aims at reducing space requirement and the other at reducing time requirement. Thus, the choice of the phrase set is about space–time tradeoff. At one extreme of the space–time tradeoff is the case of no phrases. Then all occurrences are primary and there is no secondary searching phase. The other extreme is the case of minimal number of primary occurrences, or maximal coverage of intervals with repeats. Since each string that occurs in the text must have at least one primary occurrence, a root, the following is the best we can do.

**Definition 4.14** A rooted phrase set $Z$ of a text $T$ has *maximal coverage* if every string that occurs in $T$ has exactly one primary occurrence.

There is a whole spectrum of possibilities between these extremes. A particularly interesting middle ground choice, called the Lempel–Ziv parse, is presented in Chapter 5.

Of the extremes, the case of no phrases is trivial but the maximal coverage case is not. Let us look at an example of a maximal coverage phrase set.

**Example 4.15** Let $Z$ be the set of nontrivial phrases $I \to J$ of a text $T$ such that $I$ is the first occurrence of $T(I)$ and $[\overleftarrow{J}, \overrightarrow{J}+1]$ is the first occurrence of $T[\overleftarrow{J}, \overrightarrow{J}+1]$ (or $\overrightarrow{J} = |T|$). There is also the following alternative definition of $Z$.

For $j \in [2, |T|]$, let $I_j \to J_j$ be the phrase such that $\overleftarrow{J_j} = j$, $\overrightarrow{J_j}$ is the largest integer such that $J_j$ is not the first occurrence of $T(J_j)$, and $I_j$ is the first occurrence $T(J_j)$. Then $Z = \{I_j \to J_j \mid j \in [2, |T|]\}$.

Now it is easy to see that any interval $O$ that is not the first occurrence of $T(O)$ is covered by at least one repeat, $J_{\overleftarrow{O}}$. Thus only the first occurrence of each string is a primary occurrence. Note also that the repeat covering $O$ begins at $\overleftarrow{O}$. Furthermore, for each phrase $I \to J \in Z$, $T[\overrightarrow{I}+1] \neq T[\overrightarrow{J}+1]$, and thus all phrases are right maximal.

### 4.2.2 Primary searching

The nature and difficulty of the primary search problem depends crucially on the phrase set and the pattern class. Here we look at the two extremes for the choice of the phrase set, and consider mainly the class of string patterns.

In the case of no phrases, the primary search problem is to perform a pattern matching query without any text preprocessing. The literature has a large number of such algorithms for many different pattern classes.

In the other extreme, let us consider string matching with the maximal coverage phrase set of Example 4.15. The primary search for a pattern $P$ could go as follows. Let $O_i$ be the primary occurrence of $P[1,i]$. Starting with $O_0 = [1,0]$, we use each $O_i$ to compute $O_{i+1}$ as follows:

1. If $T[\overrightarrow{O_i}+1] = P[i+1]$, then $O_{i+1} = [\overleftarrow{O_i}, \overrightarrow{O_i}+1]$.
2. Otherwise, look for the phrase $O_i \to J$ with $T[\overrightarrow{J}+1] = P[i+1]$. If such a phrase exists, then $O_{i+1} = [\overleftarrow{J}, \overrightarrow{J}+1]$.
3. Otherwise, $P[1,i+1]$ has no occurrences in $T$.

**Example 4.16** Let $T = \texttt{CABACCA}$. Then the maximal coverage phrase set of Example 4.15 is $Z = \{[1,0] \to [2,1], [1,0] \to [3,2], [2,2] \to [4,4], [1,1] \to [5,5], [1,2] \to [6,7], [2,2] \to [7,7]\}$. The primary search for the pattern $P = \texttt{CCB}$ goes as follows. In the beginning, $O_0 = [1,0]$. Since $T[1] = P[1]$, we apply step 1 and get $O_1 = [1,1]$. Now $T[2] \neq P[2]$, so we try step 2 and look for a phrase $[1,1] \to J$ with $T[\overrightarrow{J}+1] = P[2] = \texttt{C}$. The phrase $[1,1] \to [5,5]$ satisfies the conditions, so we set $O_2 = [5,6]$. Next we see that $T[7] \neq P[3]$ and there is no phrase with $[5,6]$ as the source. Thus, neither step 1 nor step 2 can be applied, and we conclude that $P$ has no occurrences in $T$.

Step 2, the only nontrivial step of the algorithm, can be implemented, for example, by storing each phrase $I \to J$ in a hash table using the pair $(I, T[\overrightarrow{J}+1])$ as the key. This data structure is, in fact, the suffix cactus, which is described in [Kär95] together with an implementation not based on hashing. Figure 4.4 shows the suffix cactus corresponding to the phrase set of Example 4.16. It also shows another suffix cactus for the same string, corresponding to a different maximal coverage phrase set with right maximal phrases.

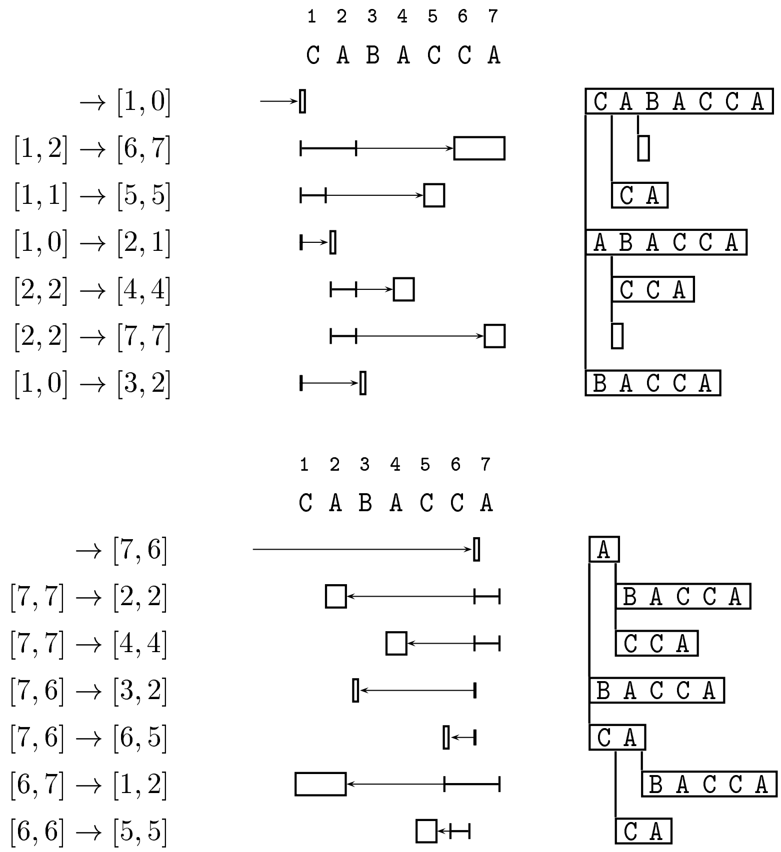



Figure 4.4: Maximal coverage phrase set and suffix cactus. On the left are two phrase sets. The first interval in each set (a repeat without a source) is the occurrence $O_0$, the starting point for the search. In the middle are visual representations of the phrase sets. On the right are the corresponding suffix cactuses.

The suffix cactus is closely related to the suffix tree [Kär95]. This can also be seen by comparing the second cactus in Figure 4.4 to the tries in Figure 3.2. Indeed, the connection between suffix cactuses and phrase sets can be extended to other compact forms of the $|\Sigma|$-ary trie as shown by the following result.

**Proposition 4.17** *The suffix tree of a text $T$ has an explicit node representing a string $S$ if and only if there exists a (possibly trivial) right maximal phrase $I \to J$ such that $T(I) = T(J) = S$. If there is only one such phrase* *and it is trivial, the node is a leaf.*

*The DAWG of a text $T$ has a node representing a string $S$ if and only if there exists a (possibly trivial) left maximal phrase $I \rightarrow J$ such that $T(I) = T(J) = S$. If at least one of the phrases is trivial, the node is a backbone node.*

*The CDAWG of a text $T$ has a node representing string $S$ if and only if there exists a (possibly trivial) maximal phrase $I \rightarrow J$ such that $T(I) = T(J) = S$. If the phrase is trivial, it is $[1, |T|] \rightarrow [1, |T|]$, and the node is the final node.*

We will not prove this result here. For a definition of suffix trees, DAWGs and CDAWGs, and a description of their properties and relationships, we refer to [BBH+87].

### 4.2.3 Secondary searching

Unlike with primary searching, the nature and difficulty of the secondary search problem are quite similar in all variations of the repetition-based indexing. In this section, we give a general solution that works for any rooted phrase set and any pattern class. It works even for the general pattern class of Definition 2.17. However, we shall also see, in this section and in later chapters, that better solutions are possible in special cases.

Below is the general algorithm for secondary searching.

**Algorithm 1** Secondary search
**Input:** rooted phrase set $Z$, set $\mathcal{O}'$ of primary occurrences
**Output:** set $\mathcal{O}$ of all occurrences
**Uses:** set $\mathcal{Q}$ of unprocessed occurrences

1 $\mathcal{O} := \mathcal{O}'$
2 $\mathcal{Q} := \mathcal{O}'$
3 **while** $\mathcal{Q} \neq \emptyset$ **do**
4     choose any $O \in \mathcal{Q}$
5     $\mathcal{Q} := \mathcal{Q} \setminus \{O\}$
6     **for each** $I \rightarrow J \in Z$ such that $O \subseteq I$ **do**
7         $O'' := [\overleftarrow{O} + \overleftarrow{J} - \overleftarrow{I}, \overrightarrow{O} + \overleftarrow{J} - \overleftarrow{I}]$
8         **if** $O'' \notin \mathcal{O}$ **then**
9             $\mathcal{O} := \mathcal{O} \cup \{O''\}$
10             $\mathcal{Q} := \mathcal{Q} \cup \{O''\}$
11 **return** $\mathcal{O}$

There are two places in Algorithm 1, where the implementation details are not trivial. The first one is on line 6, where we have to solve a range containment problem (Problem 2.31). As mentioned in Section 2.3.4, we can use any solution to the doubly semi-infinite range search problem such as the data structure mentioned in Remark 3.18.

The second nontrivial place is the check on line 8, where the purpose is to avoid processing the same occurrence multiple times. If an occurrence is contained by multiple repeats, it is found multiple times on line 6. To implement the check for duplicates, we could store the set $\mathcal{O}$ in a dynamic data structure supporting insertions and access queries. For example, the exponential search tree of Andersson [And96] supports $O(\sqrt{\log|\mathcal{O}|})$ amortized time operations.

A better way to handle duplicate occurrences is to augment the phrases with additional information about the overlaps of repeats. Let us assume that there are no redundant phrases, i.e., that no repeat is contained by another. Let $I \to J$ be a phrase and let $J'$ be the next repeat to the right of $J$. A secondary occurrence contained by $J$ is reported only if it is not contained by $J'$. Then, each secondary occurrence is reported only once. By storing $\overleftarrow{J'}$ with the phrase $I \to J$, only constant time is needed for the check. The set $\mathcal{O}$ can now be implemented, e.g., as a simple list. The one remaining inefficiency is that $O(1)$ time is still needed for each duplicate of an occurrence.

There is a way to store the phrases in such a way that the duplicates are completely avoided. Let $I$, $J$ and $J'$ be as above and let $\hat{I} = \overleftarrow{I} + \overleftarrow{J'} - \overleftarrow{J}$. Then on line 6 of Algorithm 1, we should find the phrases such that not only $O \subseteq I$ but also $\overleftarrow{O} < \hat{I}$ insuring that $O'' \not\subseteq J'$. This can also be stated as the query $(\overleftarrow{O}, \overrightarrow{O}) \in [\overleftarrow{I}, \hat{I}[ \times ]-\infty, \overrightarrow{I}]$. This kind of queries, where we ask for ranges that contain a query point, a reversal of roles from range queries, are often called *stabbing queries*. This problem could be solved using complicated data structures but we will not develop this approach further. The solution would be worthwhile only if there is a lot of overlap between the repeats, and as we have mentioned, there are reasons to avoid overlapping repeats anyway.

One phrase set that could have a lot of overlapping repeats is the maximal coverage phrase set of Example 4.15, but there are better methods for secondary searching in this special case. First, with the maximal coverage phrase set, $\hat{I} = \overleftarrow{I} + 1$ for all phrases. Thus, the condition on line 6 becomes $\overleftarrow{O} = \overleftarrow{I}$ and $\overrightarrow{O} \leq \overrightarrow{I}$, which is a much simpler query to implement. Without going into details, this approach effectively leads to making a full traversal of the subcactus of the suffix cactus from the point where the primary search ended.

But there is an even simpler method. The set of secondary occurrences of a primary occurrence $O$ is the set of all other occurrences of the string $T(O)$, and can be represented as the range of suffixes with the prefix $T(O)$. To be precise, the range also contains $O$ itself but this is no problem. The range is stored with $O$. A similar approach is not useful with most other phrase sets because, first, the secondary occurrences associated with a particular primary occurrence can not be represented as a range of suffixes, and second, the ordered array of suffixes requires more space than the rest of Algorithm 1 unless $|Z| = \Omega(|T|)$.

## 4.3 Concluding remarks

Repetition-based indexing is a general scheme for pattern matching. We have described some examples of its use in order to illustrate the wide range of ways it can be applied, and to lay the ground for what follows. In the next two chapters, we give two concrete applications of repetition-based indexing that improve on previous results on pattern matching.

# Chapter 5

# Lempel–Ziv index

In this chapter, we describe a class of phrase sets, called Lempel–Ziv (LZ) parses, lying somewhere in the middle of the space–time tradeoff spectrum, and give the associated data structures and algorithms for primary and secondary searching. The resulting solution to the string matching problem is called the Lempel–Ziv (LZ) index. The use of LZ parsing in repetition-based indexing, and indeed the whole idea of repetition-based indexing, was introduced in [KU96a]. The phrase set used here is essentially the same as in [KU96a], but the primary and secondary search algorithms are different.

## 5.1 Lempel–Ziv parse

The main idea of the class of LZ parses is to cover the text with non-overlapping repeats. We call these phrase sets Lempel–Ziv parses, because they parse the text into blocks of strings (the repeats) and because the first such parse was described by Lempel and Ziv [LZ76].

### 5.1.1 Lempel–Ziv parse and Ziv–Lempel compression

The original LZ parse was used for measuring the complexity (lack of repetition) of a text. A better known use for an LZ parse is Ziv-Lempel compression [ZL77, ZL78], where the idea is to replace the substring in a repeat with a reference to the corresponding source interval. If the space requirement of the reference is less than the space requirement of the replaced string, compression is achieved.

In decompression, the repeat substring is restored by reading the source substring. This, of course, requires that the source substring is already uncompressed. This can be guaranteed with an appropriate order of decompression provided that the LZ parse is rooted (see Example 4.6).

A large number of variations on the Ziv-Lempel compression scheme have been described. These are often divided into two classes called LZ77 and LZ78 [BWC89, BCW90], or LZ1 and LZ2 [Sto88]. The first class originates from [ZL77] and the second one from [ZL78]. The LZ77 variations replace a repeat substring with a direct reference to the source interval. The LZ78 variations replace substrings with pointers to a separate (dynamic) dictionary containing substrings or intervals.

Several algorithms for pattern matching in Ziv-Lempel compressed texts without decompression have been described [ABF96, GKPR96, KTS+98, FT98, KTSA99, NR99]. The variation of Ziv–Lempel compression in [FT98] is based on a phrase set that is essentially the same as the one described in this chapter.

### 5.1.2 Basic properties

Formally, we define LZ parse as follows.

**Definition 5.1** A *Lempel–Ziv (LZ) parse* is a phrase set that is rooted, 1-spaced and pairwise right maximal.

The terms '1-spaced' and 'pairwise right maximal' will be defined later in this section (Definitions 5.3 and 5.6). Examples of LZ parses are given in Figure 5.1 on page 66.

The definition is based on three principles, which are (in order of importance) non-overlapping repeats, maximal coverage, and minimum number of phrases. We will next take a closer look at these principles one at a time.

**Non-overlapping repeats** The key property of an LZ parse is that repeats do not overlap. In repetition-based indexing, this property has several advantages. First, coverage of intervals is not wasted by covering some intervals multiple times. Second, there can be no redundant phrases (except empty phrases). Third, we can use Algorithm 1 for secondary search with no concern about duplicate occurrences. Fourth, it is easy to recognize whether a set of phrases is rooted or not as shown by the following lemma.

**Lemma 5.2** *Let $Z$ be a set of phrases with non-overlapping repeats. The set $Z$ is rooted if and only if its phrase graph $G_Z$ is acyclic.*

**Proof** With non-overlapping repeats, each vertex of $G_Z$ can have at most one incoming edge. Thus, the connected components of $G_Z$ are either rooted trees or simple cycles. Since a cycle has no root, $Z$ is rooted if and only if there are no cycles. □

There are circumstances where limited overlap of repeats may be allowed without compromising the above benefits. For example, if it is known that no pattern will match a string shorter than $q$, then overlaps of at most $q-1$ are acceptable. Such a case is described in Chapter 6.

**Maximal coverage of text** The second principle, covering as much of the text as possible with repeats, is necessary to make primary search efficient. It is not possible to cover every character. For each different character that appears in the text, there must be at least one (a root) that is not covered by any repeat. All other characters on the other hand could be covered. In any case, there should not be large uncovered areas. The following definition captures this idea.

**Definition 5.3** A phrase set with non-overlapping repeats is *1-spaced* if there is exactly one uncovered character before the first repeat and between each pair of consecutive repeats.

**Remark 5.4** Any phrase set can be made 1-spaced by removing or shortening phrases whose repeats overlap or touch and by adding (possibly empty) phrases where necessary.

1-spacing provides non-optimal covering in two ways. First, some uncovered characters between repeats could have been covered. Second, redundant empty phrases may be needed between consecutive uncoverable characters. However, the regular spacing of repeats gives other advantages. In particular, it makes primary searching easier. In principle, some other fixed spacing could be used, but perfectly regular 0-spacing is not possible, because some characters must be left uncovered, and larger spacings would make the coverage worse.

**Minimum number of phrases** The obvious motivation for the third principle, minimum number of phrases, is to save space, but it also improves the coverage of intervals. Minimizing the number of phrases of a 1-spaced phrase set is equivalent to maximizing the average length of the phrases, which improves the coverage of long intervals.

Unfortunately, finding the rooted 1-spaced phrase set with minimal number of phrases is a hard problem. The very closely related problem of finding the rooted phrase set with non-overlapping repeats that minimizes the total number of phrases and uncovered characters has been shown to be NP-complete [Gal82].

Thus, we will not require the absolutely minimum number of phrases or maximum average length of phrases, but a weaker property based on a more local form of maximality. Note, however, that the local maximalities of Definition 4.13 would be too weak because they do not take into account the freedom in placing the sources. Instead, we use the following notion of maximality.

**Definition 5.5** Let $J$ and $J'$ be two repeats of a phrase set over a text $T$. Without loss of generality, assume that $|J| = |J'| + i$ for some $i \geq 0$. The repeats $J$ and $J'$ are *pairwise right maximal* if $[\overleftarrow{J'}, \overrightarrow{J'} + i + 1] \to [\overleftarrow{J}, \overrightarrow{J} + 1]$ is not a phrase of $T$.

Intuitively, the meaning is that we cannot make the longer repeat $J$ any longer by moving its source to start at $\overleftarrow{J'}$.

**Definition 5.6** Let $\mathcal{J}$ the set of repeats of a phrase set $Z$, and let $J_0 = [1, 0]$ be a 'dummy' repeat. The phrase set $Z$ is *pairwise right maximal* if all pairs in the set $\mathcal{J} \cup \{J_0\}$ are pairwise right maximal.

The benefits of this notion of maximality are twofold. First, it is easy to achieve with the greedy online algorithms of Section 5.1.3. Second, it is good enough for the quite strong upper bounds on the number of phrases given in Section 5.1.4.

### 5.1.3 Construction

A natural way to construct an LZ parse is a greedy online construction that scans through the text and builds the phrase set on the way. After each step of the scan, the LZ parse for the scanned part is ready. A phrase set constructed this way has the following useful property.

**Definition 5.7** A phrase $I \to J$ is *right-directional* if $I < J$. A phrase set is right-directional if all of its phrases are right-directional.

The significance of right-directionality is shown by the following result.

**Lemma 5.8** *A right-directional phrase set with non-overlapping repeats is rooted.*

**Proof** In the phrase graph of a right-directional phrase set, all edges go from the earlier interval to the later interval. Thus, the phrase graph is acyclic and, by Lemma 5.2, the phrase set is rooted. □

**Optimal construction** Below is a greedy online algorithm that constructs the minimal right-directional LZ parse.

**Algorithm 2** Greedy online construction of LZ parse
**Input:** text $T$
**Output:** right-directional LZ parse $Z$ of $T$

```
1   Z := ∅
2   i := 2
3   while i ≤ |T| + 1 do
4       Let I → J be a phrase of maximum length
                              such that ←J = i and ←I < i
5       Z := Z ∪ {I → J}
6       i := →J + 2
7   return Z
```

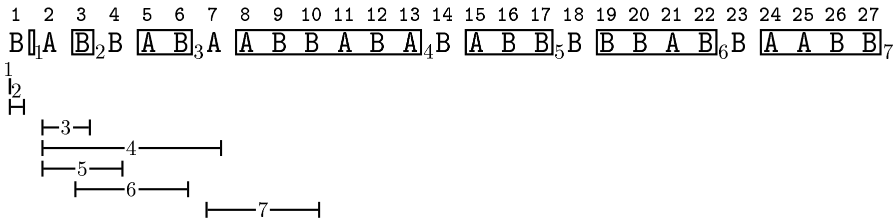
An example of an LZ parse constructed by this algorithm is in Figure 5.1(a).

Choosing the phrases in step 4 is done during an online construction of the ($|\Sigma|$-ary) suffix tree. The online construction adds suffixes to the suffix tree in left-to-right order. When adding the suffix $T[i, |T|]$, the algorithm needs to compute the longest common prefix between $T[i, |T|]$ and an earlier suffix to know where to insert $T[i, |T|]$ in the suffix tree. The common prefixes of the two suffixes form the phrase $I \to J$. There are fast online construction algorithms for the suffix tree based on the use of suffix links [McC76, Ukk95].

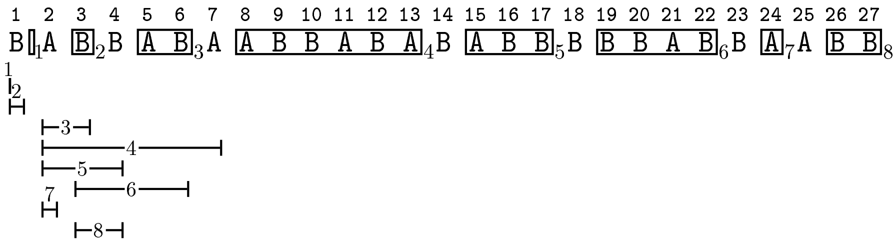
The following results show that Algorithm 2 constructs a right-directional LZ parse of minimal size.

**Lemma 5.9** *The phrase set $Z$ built by Algorithm 2 is pairwise right maximal.*

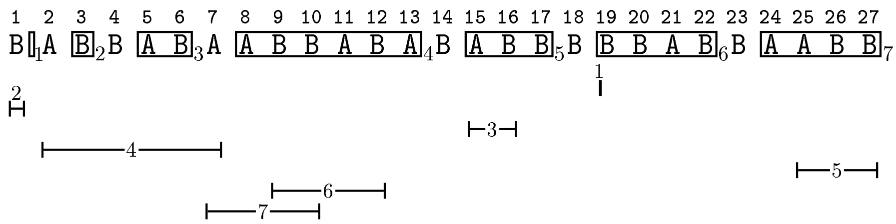
**Proof** Assume the contrary, i.e., that $[\overleftarrow{J'}, \overrightarrow{J'} + i + 1] \to [\overleftarrow{J}, \overrightarrow{J} + 1]$ is a phrase of $T$ for some repeats $J$ and $J'$ of $Z$ ($J'$ may also be the dummy repeat $[1, 0]$), and some $i \geq 0$ such that $|J| = |J'| + i$. If $J' < J$, the algorithm would not have chosen $J$ as the repeat that starts at $\overleftarrow{J}$ because $[\overleftarrow{J'}, \overrightarrow{J'} + i + 1] \to [\overleftarrow{J}, \overrightarrow{J} + 1]$ is a phrase with a longer repeat. Similarly, if $J < J'$, the repeat $J'$ would not have been chosen because $[\overleftarrow{J}, \overrightarrow{J} + 1] \to [\overleftarrow{J'}, \overrightarrow{J'} + i + 1]$ is a phrase, too. In both cases, there is a contradiction, which proves the lemma. $\square$



(a) LZ parse constructed by Algorithm 2.



(b) LZ parse constructed by the Fiala–Greene variation.



(c) LZ parse in (a) transformed into a linear LZ parse.

Figure 5.1: LZ parses for the string `BABBABAABBABABABBBBBABBAABB`.

**Theorem 5.10** *The phrase set $Z$ build by Algorithm 2 is a minimal right-directional LZ parse.*

**Proof** Clearly, $Z$ is 1-spaced, right-directional, and consequently rooted by Lemma 5.8. By Lemma 5.9, $Z$ is pairwise right maximal and is therefore an LZ parse. It remains to show that it is minimal.

Let $J_1, J_2, \ldots, J_{|Z|}$ be the repeats of $Z$ and $J'_1, J'_2, \ldots, J'_{|Z'|}$ the repeats of another right-directional LZ parse $Z'$ in left-to-right order. We prove, by

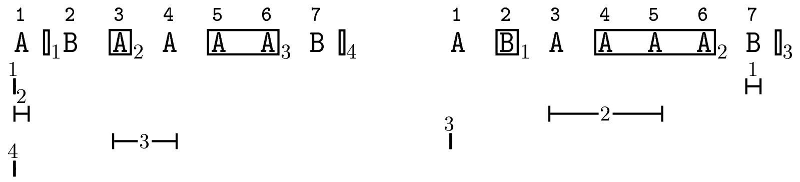



Figure 5.2: A minimal right-directional LZ parse (left) and a smaller non-right-directional LZ parse (right).

induction, that $\overleftarrow{J'_i} \le \overleftarrow{J_i}$ for all $1 \le i \le |Z|$. This shows that $|Z| \le |Z'|$. The basis $i = 1$ is clear since $\overleftarrow{J'_1} = 2 = \overleftarrow{J_1}$. Assume then that $i > 1$ and $\overleftarrow{J'_{i-1}} \le \overleftarrow{J_{i-1}}$. Then $\overrightarrow{J'_{i-1}} \le \overrightarrow{J_{i-1}}$, because $J_{i-1}$ is the longest possible repeat starting at $\overleftarrow{J_{i-1}}$. This furthermore means that $\overleftarrow{J'_i} \le \overleftarrow{J_i}$ completing the induction. □

**Remark 5.11** A non-right-directional LZ parse can be smaller than the minimal right-directional LZ parse as shown in Figure 5.2.

**Fiala–Greene construction** In [FG89], Fiala and Greene describe a Ziv-Lempel compression scheme, which is one of the best variations of the family [BCW90]. The essence of their parsing algorithm is to replace the line 4 of Algorithm 2 with

4' Let $I \to J$ be a phrase of maximum length such that
$$\overleftarrow{J} = i \text{ and } \overleftarrow{I} \in \{\overleftarrow{J'} \mid I' \to J' \in Z\} \cup \{1\}$$

In other words, there is the additional restriction that the next source must start at the beginning of a previous repeat or at the beginning of the text. The change is implemented by replacing the suffix tree with a sparse suffix tree containing only the relevant suffixes. An example of the Fiala–Greene LZ parse is shown in Figure 5.1(b).

The resulting LZ parse might not be minimal but it is still quite small as shown by the compression results of the Fiala–Greene algorithm [BCW90] and by the analysis in the next section. The advantage over the optimal version is a simpler construction algorithm that takes less space. While Algorithm 2 needs $\Omega(|T| \log |T|)$ bits for the suffix tree, the Fiala–Greene version needs only $O(|Z| \log |T|)$ bits for the sparse suffix tree.

**Theorem 5.12** *The phrase set $Z$ constructed by Fiala–Greene parsing is an LZ parse.*

**Proof** Clearly, $Z$ is 1-spaced, right-directional, and thus rooted. What remains to show is that $Z$ is pairwise right maximal. This is shown by the proof of Lemma 5.9, which applies here, too, word for word. □

**Linear LZ parse** Linearity of an LZ parse is a helpful property for some secondary searching algorithms.

**Definition 5.13** An LZ parse $Z$ is *linear* if no source contains another source.

Linearity means that the set of sources is fully $\leq$-ordered (Lemma 2.11).

**Lemma 5.14** *Any LZ parse can be transformed into a linear LZ parse with the same set of repeats.*

**Proof** Let $I \rightarrow J$ and $I' \rightarrow J'$ be phrases of an LZ parse $Z$ such that $I$ contains $I'$. The violation of linearity can be removed by replacing $I'$ with $I'' = [\overleftarrow{I'} + \overleftarrow{J} - \overleftarrow{I}, \overrightarrow{I'} + \overleftarrow{J} - \overleftarrow{I}]$, i.e., by moving $I'$ from inside $I$ to inside $J$. Because $I$ and $J$ have the same content, $I'' \rightarrow J'$ is still a phrase. The resulting phrase set $Z'$ is 1-spaced and is pairwise right maximal because the repeats have not changed.

The above transformation is repeated until there are no more violations of linearity. We still have to show that the transformation preserves rootedness and that the process ends eventually. We will do this by considering the changes in the phrase graph.

Let $O$ be an interval contained by $I'$. In the phrase graph $G_Z$, $O$ has at least two children, one (call it $V'$) inside $J'$, and another (call it $V$) inside $J$. As a result of the above transformation, the edge from $O$ to $V'$ is replaced by the edge from $V$ to $V'$. In other words, the start of the edge is moved from a parent to a child. All changes in $G_Z$ are of this type. Thus the phrase graph will stay a forest and, by Lemma 5.2, $Z'$ is rooted. Furthermore, the changes move edges downwards in the trees and thus the process must end at latest when all trees are linear lists. □

**Remark 5.15** The result of the transformation is likely to lose right-directionality.

An example of a linear LZ parse is shown in Figure 5.1(c).

### 5.1.4 Analysis

We will now analyze the number of phrases in an LZ parse. The space requirement of the LZ index has at least linear dependence on it.

For the rest of this subsection, let $T$ be a text, and $Z$ an LZ parse of $T$. Let $||Z||$ denote the sum of the lengths of the phrases of $Z$. Let $c_l$ denote the number of distinct substrings of length $l$ in $T$, and $c = c_1$ the number of different characters in $T$. Let $N = |Z|$. For simplicity, assume that $|T| > c \geq 2$; the other cases are trivial. Then $N \geq 1$.

**Lemma 5.16** *Let $l \geq 0$. The LZ parse $Z$ has at most $c_{l+1}$ phrases of length at most $l$.*

**Proof** Let $I \to J$ and $I' \to J'$ be two different phrases of length at most $l$. Append $l+1$ arbitrary characters to the end of the text so that $T[\overleftarrow{J}, \overleftarrow{J}+l]$ and $T[\overleftarrow{J'}, \overleftarrow{J'}+l]$ are certain to be defined. By the pairwise right maximality, $T[\overleftarrow{J}, \overleftarrow{J}+l] \neq T[\overleftarrow{J'}, \overleftarrow{J'}+l]$ except if $J$ or $J'$ is the last phrase of the text, and furthermore, $T[\overleftarrow{J}, \overleftarrow{J}+l] \neq T[1, 1+l] \neq T[\overleftarrow{J'}, \overleftarrow{J'}+l]$. Thus, the number of such phrases cannot be more than $c_{l+1}$. □

In particular, the number of empty phrases in $Z$ is at most $c$.

**Lemma 5.17** *Let $\ell$ be the integer such that $c_\ell < N \leq c_{\ell+1}$. The sum of the lengths of the phrases is $||Z|| \geq N\ell - \sum_{i=1}^{\ell} c_i$.*

**Proof** By Lemma 5.16, an LZ parse with the smallest combined length would have $c_1$ empty phrases, $c_2 - c_1$ phrases of length 1, $c_3 - c_2$ phrases of length 2, etc. The longest phrases would be of length $\ell$ and their number would be $N - c_\ell$. Thus a lower bound for the sum of the lengths is

$$\sum_{i=1}^{\ell-1} i\Big(c_{i+1} - c_i\Big) + \ell(N - c_\ell) = N\ell - \sum_{i=1}^{\ell} c_i.$$

□

For the next step, we need the following result.

**Lemma 5.18** *For any $0 \leq b \leq 1$ and $c \geq 1$, $b(c-1) \geq c^b - 1$.*

**Proof** Let $f(c) = b(c-1) - (c^b - 1)$. Since $f'(c) = b - bc^{b-1} \geq 0$ everywhere in the range, $f(c) \geq f(1) = 0$. □

**Lemma 5.19** *The length of the text $T$ is*

$$|T| \geq N\left(\log_c N\right) - \frac{N-c}{c-1}.$$

**Proof** Due to 1-spacing, the length of the text is

$$|T| = ||Z|| + N \geq N(\ell+1) - \sum_{i=1}^{\ell} c_i.$$

The lower bound is minimal when each $c_i$ is as large as possible. Thus, assume that $c_i = c^i$ for all $i$. Then $\ell + 1 = \lceil \log_c N \rceil = \log_c N + b$, where $b = \lceil \log_c N \rceil - \log_c N$. Thus

$$\begin{aligned}
|T| &\geq N(\ell+1) - \frac{c^{\ell+1}-c}{c-1} \geq N\left(\log_c N + b\right) - \frac{Nc^b}{c-1} + \frac{c}{c-1} \\
&\geq N\left(\log_c N + b - \frac{c^b}{c-1}\right) + \frac{c}{c-1} \geq N\left(\log_c N - \frac{1}{c-1}\right) + \frac{c}{c-1} \\
&= N\left(\log_c N\right) - \frac{N-c}{c-1},
\end{aligned}$$

because $b - c^b/(c-1) \geq -1/(c-1)$ by lemma 5.18. □

**Theorem 5.20** *The size of the LZ parse $Z$ is at most*

$$|Z| \leq \frac{|T| - c/(c-1)}{\log_c |T| - \log_c \log_c |T| - 1/(c-1)}.$$

**Proof** Let $f(x) = x \log_c x - (x-c)/(c-1)$. It is easy to verify by elementary calculus that $f(x)$ is an increasing function of $x$ when $c \geq 2$ and $x \geq 1$. Also, easy to verify is that $f(n/\log_c n) \leq n$ when $n > c \geq 2$. Let $\widehat{N}$ be such that $f(\widehat{N}) = |T|$. Then $\widehat{N} \geq |T|/\log_c |T|$, and

$$\begin{aligned}
|T| &= \widehat{N}\left(\log_c \widehat{N} - \frac{1}{c-1}\right) + \frac{c}{c-1} \\
&\geq \hat{N}\left(\log_c |T| - \log_c \log_c |T| - \frac{1}{c-1}\right) + \frac{c}{c-1}.
\end{aligned}$$

By Lemma 5.19, $f(N) \leq |T| = f(\widehat{N})$, and therefore

$$N \leq \hat{N} \leq \frac{|T| - c/(c-1)}{\log_c |T| - \log_c \log_c |T| - 1/(c-1)}.$$

□

We have shown that $|Z| = O(|T|/\log_{|\Sigma|}|T|)$. This is, on the other hand, also the asymptotically minimal size of an LZ parse in the worst case as shown by the following result.

**Theorem 5.21** *For any positive integer $n$ and any alphabet $\Sigma$, there exists a text $T \in \Sigma^n$ such that any LZ parse $Z$ of $T$ satisfies*

$$|Z| \geq \frac{|T|}{\left\lceil \log_{|\Sigma|}|T| \right\rceil}.$$

**Proof** For any positive integer $k$ and an alphabet $\Sigma$ there exists a string of length $|\Sigma|^k + k - 1$ which contains exactly one occurrence of each string in $\Sigma^k$. Such strings, known as de Bruijn sequences, can have no nontrivial phrases longer than $k-1$. Let $k$ be the smallest integer such that $|\Sigma|^k + k - 1 \geq n$, which means that $k = \lceil \log_{|\Sigma|}(n-k+1) \rceil \leq \lceil \log_{|\Sigma|} n \rceil$. Let $T$ be the prefix of length $n$ of the de Bruijn sequence of length $|\Sigma|^k + k - 1$. Then, any LZ parse over $T$ must have at least $n/k \geq n/\lceil \log_{|\Sigma|} n \rceil$ phrases. □

Both the upper and the lower bound hold for any LZ parse. Similar upper and lower bounds were given for the minimal right-directional LZ parse in [LZ76].

In the LZ77 variation of Ziv–Lempel compression, a repeat is replaced with a reference to a source. Such a reference requires $\Theta(\log|T|)$ bits (on average).[1] Thus, the size of the compressed text is $\Theta(|Z|\log|T|)$ bits which, by the above results, is $O(|T|\log|\Sigma|)$ bits, the same as the size of the original text. However, this is for the worst case, an incompressible text. If the text is compressible by Ziv-Lempel compression, the compression is achieved by reducing the number of phrases. Then the space requirement of the LZ index is reduced, too.

## 5.2 Algorithms

Next we describe primary and secondary searching algorithms for string pattern matching with an LZ parse.

[1] This assumes that the position of sources is not restricted, e.g., to be inside a smaller window as is done in many variations. But in such a case, the upper bound on the number of phrases would not hold either.

### 5.2.1 Primary searching

The primary search problem is to find the primary occurrences of a pattern $P$, i.e., the occurrences that are not contained by any repeat. Primary occurrences can be alternatively characterized as follows.

**Definition 5.22** A *boundary* of a phrase set is an interval that is not contained by any repeat of the phrase set but whose every proper subinterval is contained by some repeat.

**Lemma 5.23** *An occurrence is primary if and only if it contains a boundary.*

This characterization, although applicable to any phrase set, is particularly helpful for LZ parses as shown by the following result.

**Lemma 5.24** *Let $J_1 < J_2 < \cdots < J_N$ be the phrase repeats of an LZ parse $Z$. The boundaries of $Z$ are the intervals $[1,0]$ and $[\overleftarrow{J_i} - 1]$, for $i = 2, \ldots, N$.*

**Remark 5.25** The interval $[1,0]$ is the only primary occurrence of the empty interval. Any nonempty interval that contains $[1,0]$ must also contain $[1] = [\overleftarrow{J_1} - 1]$. Thus, assuming an empty interval cannot be an occurrence, we can use $[\overleftarrow{J_i} - 1]$, for $i = 1, \ldots, N$, as the set of boundaries.

In other words, to check whether an occurrence is primary, all we have to know about the LZ parse is the positions (intervals of length 1) of the $N$ boundaries.

With a string pattern $P$, the length of all occurrences is $|P|$. Thus, every primary occurrence $O$ of $P$ can be specified by giving a boundary that $O$ contains and the position of $O$ that overlaps the boundary.

**Definition 5.26** Let $Z$ be an LZ parse of a text $T$, and $B = \{[b_1], \ldots, [b_N]\}$, $b_1 < b_2 < \cdots < b_N$ the boundaries of $Z$. Also, define $b_0 = 0$. Let $O$ be a primary occurrence of a string pattern $P$ in $T$. A *reference pair* of $O$ is a pair $(i,j)$, $1 \leq i \leq N$, $1 \leq j \leq |P|$, such that $[b_i - j + 1, b_i - j + |P|] = O$. If, additionally, $b_i - j + 1 > b_{i-1}$, the pair $(i,j)$ is the *canonical* reference pair and $[b_i]$ the *canonical boundary* of $O$.

The canonical boundary is simply the left-most one of the boundaries of an occurrence.

There is a one-to-one correspondence between canonical reference pairs and primary occurrences. Thus, the primary search problem is to find all pairs $(i,j)$, $1 \leq i \leq N$, $1 \leq j \leq |P|$, satisfying the following conditions:

1. $T[b_i - j + 1, b_i - j + |P|] = P$ (specifies an occurrence)

2. $b_i - j + 1 > b_{i-1}$ (is canonical)

For each such pair, $[b_i - j + 1, b_i - j + |P|]$ is a distinct primary occurrence.

### 5.2.2 Reduction to two-dimensional range searching

It is helpful to break a primary occurrence into three parts: the canonical boundary and the parts before and after the boundary (which may be empty). Then we can rewrite the conditions that a canonical reference pair $(i, j)$ must satisfy as follows:

1. $P[j] = T[b_i]$

2. $P[1, j-1]$ is a suffix of $T[b_{i-1} + 1, b_i - 1]$ and

3. $P[j + 1, |P|]$ is a prefix of $T[b_i + 1, |T|]$.

Note the asymmetry in the last two conditions; the middle one includes the canonicality condition.

For each $a \in \Sigma$, let $B_a$ be the set $\{b_i \mid 1 \leq i \leq N,\ T[b_i] = a\}$. The following algorithm computes the primary occurrences.

**Algorithm 3** Primary search

1 $\mathcal{O}' := \emptyset$
2 **for** $j \in [1, |P|]$ **do**
3   $a := P[j]$
4   $X = \{b_i \in B_a \mid P[1, j-1]$ is a suffix of $T[b_{i-1} + 1, b_i - 1]$
      and $P[j+1, |P|]$ is a prefix of $T[b_i + 1, |T|]\}$
5   $\mathcal{O}' := \mathcal{O}' \cup \{[b_i - j + 1, b_i - j + |P|] \mid b_i \in X\}$
6 return $\mathcal{O}'$

Computing $X$ on line 4 is the difficult part of the algorithm. We need, for each $B_a$, a data structure that can be used for quickly computing $X$ given the strings $P[1, j-1]$ and $P[j+1, |P|]$. We can break the task into subtasks as follows.

4.1 $X_1 = \{b_i \in B_a \mid P[1, j-1]$ is a suffix of $T[b_{i-1} + 1, b_i - 1]\}$
4.2 $X_2 = \{b_i \in B_a \mid P[j+1, |P|]$ is a prefix of $T[b_i + 1, |T|]\}$
4.3 $X = X_1 \cap X_2$

The computation of $X_2$ is a prefix matching query (Problem 2.24). The computation of $X_1$ is the symmetric suffix matching query. Thus $X_1$ and $X_2$ are ranges in two different orderings of $B_a$. We can think of a boundary $b_i \in B_a$ as a reference to the point $(T[b_{i-1}+1, b_i-1], T[b_i+1, |T|])$ of the two-dimensional universe $\Sigma^* \times \Sigma^*$. Then $X_1$ and $X_2$ are ranges in the two dimensions, $X$ is a two-dimensional range, and the computation on line 4 is a two-dimensional range query.

In Section 3.6, we gave solutions to two-dimensional range searching in *rank space*. For two-dimensional range searching in $\Sigma^* \times \Sigma^*$, we also need a transformation to the rank space. In the above algorithm, the sets $X_1$ and $X_2$ can be represented as ranges of ranks. Then lines 4.1 and 4.2 perform the transformations into the rank space and can be implemented with data structures for ranking strings. Line 4.3 performs the two-dimensional range query in the rank space.

Now we have reduced primary searching to string ranking and two-dimensional range searching in rank space. Algorithm 3 does $O(|P|)$ queries of each type, and these queries dominate the time requirement of the algorithm. We need 3 data structures for each of the boundary sets $B_a$, $a \in \Sigma$, two for ranking and one for range searching. The total number of points in these data structures is $\Theta(|Z|)$. The worst case for both space and time is when one set $B_a$ has $\Omega(|Z|)$ boundaries, and that set is queried $\Omega(|P|)$ times. This is the case we analyze below.

### 5.2.3 Secondary searching

Before the analysis of primary searching, we take a look at secondary searching, which is much easier. In Section 4.2.3, we described Algorithm 1, a general algorithm for secondary searching. Algorithm 1 has two places where implementation details are nontrivial, the range containment query on line 6 and the duplicate check on line 8. With LZ parses, the latter is not a problem: with non-overlapping repeats, no occurrence is found multiple times.

The range containment query could be implemented with the data structure of Remark 3.18 as suggested in Section 4.2.3. The resulting time for the whole secondary search is $O(k \log\log |T|)$.

In Lemma 5.14, we showed that any LZ parse can be made linear without affecting the repeats and, thus, primary searching. Thus we can assume linearity, which enables better methods for the range containment query. In a linear LZ parse, the sources are fully $\leq$-ordered. The sources that contain an occurrence $O$ form a range in this ordering. The range can be computed with two ranking queries, one to determine the first source $I$ with $\overrightarrow{I} \geq \overrightarrow{O}$, and the other to determine the last source $I'$ with $\overleftarrow{I'} \leq \overleftarrow{O}$.

Table 5.1: Space and time requirements of the LZ index with different solutions for the two-dimensional range search problem.

| space | query time |
|---|---|
| $\|T\|\frac{\log\|T\|\log\|\Sigma\|}{\log\log\log\|T\|}$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\log\log\|T\| + k$ |
| $\|T\|\frac{1}{\varepsilon}\frac{\log\|T\|\log\|\Sigma\|}{\log\log\|T\|}$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\log^{\varepsilon}\|T\| + k$ |
| $\|T\|\left(\log\|\Sigma\| + \frac{1}{\varepsilon}\log^{\varepsilon}\|T\|\right)$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\log\|T\| + \frac{1}{\varepsilon}k$ |
| $\|T\|\left(\log\|\Sigma\| + \log\log\|T\|\right)$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\log\|T\| + k\log\log\|T\|$ |
| $\|T\|\left(\log\|\Sigma\| + \frac{1}{\varepsilon}\right)$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\log\|T\| + \frac{1}{\varepsilon}k\log^{\varepsilon}\|T\|$ |
| $\|T\|\left(\log\|\Sigma\| + \frac{1}{\varepsilon}\right)$ | $\frac{\|P\|^2\log\|\Sigma\|}{\log\|T\|} + \|P\|\|T\|^{\varepsilon} + k$ |

For the ranking, we use the bucketed bitmap of Theorem 3.1. The space requirement is $O(|T|)$ bits since the universe is $[1, |T|]$. With the $O(1)$ query time of the bucketed bitmap, the whole secondary search time is $O(1 + k)$.

### 5.2.4 Analysis

With the efficient method for secondary searching, the time and space requirement of the whole string matching query is dominated by primary searching. The primary searching, in turn, is dominated, in the worst case, by $O(|P|)$ ranking queries in a set of $O(|Z|)$ strings and $O(|P|)$ two-dimensional range queries in the rank space of a set of $O(|Z|)$ points.

For string ranking, we use the path-compressed trie of Theorem 3.11, and in particular, the variation of Table 3.3 (page 36). The assumptions of the table, that the space requirement of the task is polynomial in the length of the text and that the query time is optimized for long patterns, are good for the present case. Since now $|V| = O(|Z|) = O(|T|/\log_{|\Sigma|}|T|)$, the space requirement is $O(|T|\log|\Sigma|)$ bits. There are $O(|P|)$ queries requiring $O(|P|\log|\Sigma|/\log|T| + \log\log|T|)$ time each, which gives a total time of $O(|P|^2\log|\Sigma|/\log|T| + |P|\log\log|T|)$.

For the remaining problem of two-dimensional range searching, we can use any of the data structures of Section 3.6. The properties of a number of data structures with different space–time tradeoffs were given in Table 3.7. In the present case, $|V| = |Z|$ and each string matching query needs $O(|P|)$ two-dimensional range queries. Assuming $|Z| = \Theta(|T|/\log_{|\Sigma|}|T|)$ (the worst case), the properties of the whole LZ index corresponding to the different solutions for two-dimensional range searching are given in Table 5.1.

We are particularly interested in solutions with the optimal space requirement of $O(|T| \log |\Sigma|)$ bits. In Table 5.1, the value $\varepsilon$ is shown explicitly everywhere. Thus it does not need to be a constant. On the last two lines, we can replace $\varepsilon$ with $\varepsilon / \log |\Sigma|$ to get the following theorem.

**Theorem 5.27** *A text $T \in \Sigma^*$ can be stored in $O(\frac{1}{\varepsilon}|T| \log |\Sigma|)$ bits so that all occurrences of a pattern $P$ can be found in time*

$$O\left(\frac{|P|^2 \log |\Sigma|}{\log |T|} + |P| \log |T| + \frac{1}{\varepsilon} k \log |\Sigma| \log^{\varepsilon / \log |\Sigma|} |T|\right)$$

*or*

$$O\left(\frac{|P|^2 \log |\Sigma|}{\log |T|} + |P||T|^{\varepsilon / \log |\Sigma|} + k\right),$$

*where $k$ is the number of occurrences and $\varepsilon$ is a positive constant.*

Of the other possibilities, the following two are particularly interesting.

**Theorem 5.28** *A text $T \in \Sigma^*$ can be stored in $O(|T|(\log |\Sigma| + \log \log |T|))$ bits so that all occurrences of a pattern $P$ can be found in time*

$$O\left(\frac{|P|^2 \log |\Sigma|}{\log |T|} + |P| \log |T| + k \log \log |T|\right),$$

*where $k$ is the number of occurrences.*

**Theorem 5.29** *A text $T \in \Sigma^*$ can be stored in $O(|T|(\log |\Sigma| + \frac{1}{\varepsilon} \log^{\varepsilon} |T|))$ bits so that all occurrences of a pattern $P$ can be found in time*

$$O\left(\frac{|P|^2 \log |\Sigma|}{\log |T|} + |P| \log |T| + \frac{1}{\varepsilon} k\right),$$

*where $k$ is the number of occurrences and $\varepsilon$ is a positive constant.*

The space and time requirements of the different variations of the LZ index are generally somewhere between the traditional text indexes and sequential search algorithms. The interesting point of comparison is the sparse suffix tree (Section 3.4.2). Comparing Table 5.1 to Table 3.6 shows that the properties of the two data structures are remarkably similar. The LZ index has the extra term $O(|P|^2 \log |\Sigma| / \log |T|)$ in all time requirements, but if that term is ignored, the LZ index dominates the sparse suffix tree in all areas of the space–time tradeoff spectrum. In particular, the results in Theorems 5.27, 5.28 and 5.29 cannot be matched by the sparse suffix tree.

## 5.3 Concluding remarks

The different space–time tradeoffs were achieved by using different methods for the two-dimensional range searching. Another possibility for adjusting the tradeoff would be to change the phrase set. In particular, the spacing of repeats could be either larger or smaller, including letting the repeats overlap (see next chapter). The primary searching scheme described here can, in principle, be extended for such cases. The concept of a boundary (Definition 5.22) was already defined for arbitrary phrase sets. Based on Lemma 5.23, boundaries can be used as 'anchor points' for primary searching. As an exercise, the reader may think what boundaries are for the maximal coverage phrase sets of Example 4.15.

An interesting area of the space–time plane is the space of $o(|T|\log|\Sigma|)$ bits. In [FT98], Farach and Thorup describe an algorithm for (unindexed) string matching in Ziv–Lempel compressed texts without decompression. The compression is based on a phrase set that is essentially an LZ parse. The space requirement is $O(|Z|\log|T|)$ bits, the size of the compressed text (see comments in Section 5.1.4). This is $o(|T|\log|\Sigma|)$ bits if the text compresses enough. The search requires $O(|Z|\log^2(|T|/|Z|)+|P|)$ randomized time. In LZ indexing, most of the data structures can be fit in $O(|Z|\log|T|)$ bits. The exception is the text itself; the uncompressed text is needed in string ranking. To get to $o(|T|\log|\Sigma|)$ bits, we need a data structure for string ranking that works over a compressed text. One possible approach is described in [dMNZ97]. It uses (sparse) suffix arrays on a text compressed with word-based Huffman coding [BSTW86, Mof89]. The method is intended for natural language texts, but could perhaps be adapted to work with LZ indexing. Alternatively, it might be possible to replace the suffix array with a repetition-based index using the repetition information in the Huffman coding.

# Chapter 6

# Lempel–Ziv index for $q$-grams

Finding all occurrences of given $q$-grams is an important subproblem in some approximate string matching algorithms. In this chapter, we describe a new repetition-based text index for $q$-gram matching that uses a variation of the Lempel–Ziv parse as the phrase set. The material is mostly from [KS98].

## 6.1 $q$-gram indexes

A $q$-gram index of a text is a data structure that allows one to quickly find all occurrences of a given $q$-gram in the text. In other words, it is a pattern matching index for the pattern class of $q$-gram patterns (Definition 2.19).

### 6.1.1 Approximate string matching with $q$-grams

The approximate string matching query asks for all occurrences of an approximate string pattern (Definition 2.20) in a text. A large number of methods for the problem have been developed (see, e.g., [Nav98]). Many of the best algorithms work in two phases. The first phase is a filtering phase that identifies a set of text regions where all the occurrences must be. The second phase then scans those regions with a sequential algorithm to find the occurrences.

Many different ways for characterizing the interesting text regions have been described. Several of them are based on $q$-grams. The idea is that an interesting region must have some amount of $q$-grams common with the pattern. There are many variations of the methods. One way to classify them is by what $q$-grams are involved: 1) all pattern and text $q$-grams [JU91, HS94], 2) all text $q$-grams and a set of non-overlapping pattern $q$-grams [Mye94, NBY98], or 3) all pattern $q$-grams and a set of non-overlapping text $q$-grams [ST96]. All the mentioned references use a $q$-gram index to quickly find the matching text $q$-grams; there are also many online or unindexed $q$-gram methods that find the text $q$-grams by a sequential search. A good exposition of both indexed and unindexed $q$-gram methods is given in [Sut98].

Of the three categories mentioned above, the last one is not relevant to us because the $q$-gram index developed here represents all text $q$-grams. The algorithms in the second category actually use non-overlapping pattern substrings that need not be $q$-grams. In fact, Myers' algorithm [Mye94] generates all approximate variations of the pattern substrings. However, both in [Mye94] and in [NBY98], a $q$-gram index for the text is used, and it can be replaced with the LZ index for $q$-grams.

### 6.1.2 Traditional $q$-gram indexes

The traditional implementation of a $q$-gram index is an access structure, such as a trie, a lookup table or a hash table, that provides an access to a list of occurrences of the query $q$-gram. The space requirement of such a data structure is $O(|T| \log |T|)$ bits and query time is $O(1 + k)$, where $k$ is the number of occurrences. We assume here, and throughout the chapter, that a $q$-gram fits into a constant number of words, and that the query $q$-gram is given in bit-packed form; otherwise the the time to process the query $q$-gram, e.g. $\Theta(q)$, should be added to the query time.

The query time of the traditional $q$-gram index is obviously optimal, but the space requirement is not. When $q$ is small enough, the access structure can be quite small, but $O(|T| \log |T|)$ bits is always required for the occurrence lists. The repetition-based $q$-gram index described here also uses an access method for the distinct $q$-grams in the primary search phase. Any access method of a traditional index can be used. The lists of occurrences, however, are replaced by a new secondary search data structure that has a smaller space requirement but still offers optimal query time.

## 6.2 Lempel–Ziv parse for $q$-grams

The phrase set we use combines the benefits of two phrase sets presented in earlier chapters, the maximal coverage phrase sets (Definition 4.14) and LZ parses (Definition 5.1). It does not satisfy the letter of either definition, but it does satisfy the spirit of both. The phrase set is like a maximal coverage phrase set except that instead of covering all but one occurrence of each substring, it covers all but one occurrence of each $q$-gram. It is like an LZ parse except that repeats are allowed to overlap by $q - 1$ positions. Then, no occurrence of a $q$-gram is covered more than once and the phrase set has all the benefits of non-overlapping repeats. We will call the phrase set an LZ parse for $q$-grams.

### 6.2.1 Construction

The following algorithm computes the LZ parse for $q$-grams and also all the primary occurrences of the $q$-grams. An example of the operation of the algorithm is shown in Figure 6.1.

**Algorithm 4** Construction of the LZ parse for $q$-grams
**Input:** text $T$
**Output:** LZ parse $Z$ for $q$-grams,
set $\mathcal{Q}$ of primary occurrences of $q$-grams

1 $Z := \emptyset$
2 $\mathcal{Q} := \{[1, q]\}$
3 $i := 2$
4 **while** $i \leq |T| - q + 1$ **do**
5 Let $I \to J$ be the phrase with smallest $\overleftarrow{I}$ such that
$\overleftarrow{J} = i$, $\overleftarrow{I} < i$ and the phrase length is maximal
6 **if** $|J| \geq q$ **then**
7 $Z := Z \cup \{I \to J\}$
8 $i := \overrightarrow{J} - q + 2$
9 **else**
10 $\mathcal{Q} := \mathcal{Q} \cup \{[i, i + q - 1]\}$
11 $i := i + 1$
12 **return** $Z$, $\mathcal{Q}$

The algorithm is quite similar to Algorithm 2 for the basic LZ parse construction. In fact, for $q = 0$ it is the same algorithm except for the requirement on line 5 that $\overleftarrow{I}$ is as small as possible. Algorithm 2 allows any choice of $\overleftarrow{I}$ that satisfies the other requirements.

The implementation details are the same, including choosing the phrases on line 5 during an online construction of the suffix tree. The additional requirement of a minimal $\overleftarrow{I}$ is naturally satisfied by the construction. As with Algorithm 2, we can have the Fiala–Greene variation by replacing line 5 with

5' Let $I \to J$ be the phrase with smallest $\overleftarrow{I}$ such that
$\overleftarrow{J} = i$, $\overleftarrow{I} \in \{\overleftarrow{J'} \mid I' \to J' \in Z \text{ or } J' \in \mathcal{Q}\}$
and the phrase length is maximal

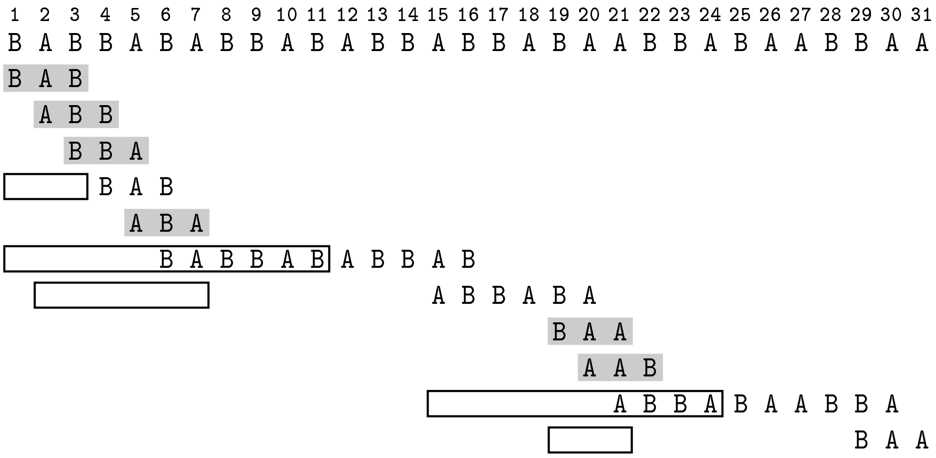



Figure 6.1: An example of LZ parsing for 3-grams by Algorithm 4. The text is at the top. The rest of the lines show the repeats (white background) and the primary occurrences of 3-grams (gray background) in the order they are formed by Algorithm 4. The sources are shown as boxes on the line of the corresponding repeats. The output of Algorithm 4 in this case is $\{[1,3] \rightarrow [4,6], [1,11] \rightarrow [6,16], [2,7] \rightarrow [15,20], [15,24] \rightarrow [21,30], [19,21] \rightarrow [29,31]\}, \{[1,3], [2,4], [3,5], [5,7], [19,21], [20,22]\}$.

Note again that, if $q = 0$, this is identical to the algorithm in Section 5.1.3 with the exception of the requirement that $\overleftarrow{I}$ is minimal.

The following lemma is a result of the additional requirement that $\overleftarrow{I}$ is minimal.

**Lemma 6.1** *No repeat of a phrase set $Z$ constructed by Algorithm 4 contains a source of $Z$.*

**Proof** By the requirement that $\overleftarrow{I}$ is as small as possible, a source $I$ is the first occurrence of $T(I)$ in $T$. On the other hand, a repeat $J'$ is never the first occurrence of $T(J')$. Thus a repeat $J'$ cannot contain a source $I$. $\square$

**Remark 6.2** Any LZ parse could be made to satisfy Lemma 6.1 by moving sources but not repeats. The idea is similar to the linearization in Lemma 5.14.

The phrase set constructed by Algorithm 4 (and its Fiala–Greene variation) also has the key property of LZ parses, pairwise right maximality

(Definition 5.6) and, in fact, a slightly stronger property as shown by the following lemma.

**Lemma 6.3** *Let $Z$ be the phrase set constructed by Algorithm 4, and let $Z'$ consist of $[1,0] \to [1,0]$ and the set of phrases that are constructed on line 5 but discarded on line 6 because they are shorter than $q$. The set $Z \cup Z'$ is pairwise right maximal.*

**Proof** The proof is the same as for Lemma 5.9. □

Algorithm 4 also computes the set $\mathcal{Q}$ of $q$-gram occurrences with the following characterizations.

**Lemma 6.4** *Let $Z$ and $\mathcal{Q}$ be the outputs of Algorithm 4. The set $\mathcal{Q}$ is the set of primary occurrences of $q$-grams with respect to the phrase set $Z$, and the set of first occurrences of $q$-grams in the text $T$.*

**Proof** Consecutive repeats constructed by Algorithm 4 overlap by $q-1$ positions except when a phrase is discarded on line 6. Thus the repeats cover all $q$-grams except those added to the set $\mathcal{Q}$. Therefore, $\mathcal{Q}$ is the set of primary occurrences. Since the phrases are made as long as possible on line 5, the discarding on line 6 happens if and only if the $q$-gram left uncovered has not occurred before. Therefore, $\mathcal{Q}$ is the set of first occurrences. □

### 6.2.2 Analysis

We will now analyze the number of phrases in an LZ parse for $q$-grams. It is useful to compare the analysis to the analysis in Section 5.1.4, especially in the case $q = 0$.

For the rest of this subsection, let $T$ be a text, and $Z$ and $\mathcal{Q}$ the output of Algorithm 4. Let $||Z||$ denote the sum of the lengths of the phrases. Let $c_l$ denote the number of distinct substrings of length $l$ in $T$, and $c = c_1$ the number of different characters in $T$. Let $N = |Z|$, and $M = |\mathcal{Q}| = c_q$. For simplicity, assume that $|T| > c \geq 2$; the other cases are trivial.

**Lemma 6.5** *Let $l \geq q$. Then $Z$ has at most $c_{l+1} - c_q + 1$ phrases of length at most $l$.*

**Proof** Let $Z'$ be as in Lemma 6.3. Let $I \to J$ and $I' \to J'$ be two phrases of $Z \cup Z'$ of length at most $l$. Append $l - q + 1$ arbitrary characters to the end of the text so that $T[\overleftarrow{J}, \overleftarrow{J} + l]$ and $T[\overleftarrow{J'}, \overleftarrow{J'} + l]$ are certain to be defined. By Lemma 6.3, $T[\overleftarrow{J}, \overleftarrow{J}+l] \neq T[\overleftarrow{J'}, \overleftarrow{J'}+l]$ except if $J$ or $J'$ is the last phrase of the text. Thus, the number of such phrases cannot be more than $c_{l+1}+1$. Of those phrases $|Z'| = |\mathcal{Q}| = c_q$ are not in $Z$. □

**Lemma 6.6** *Let $\ell$ be the integer such that $c_\ell - c_q + 1 < N \leq c_{\ell+1} - c_q + 1$. The sum of the lengths of the phrases is $||Z|| \geq (N+M-1)\ell - Mq - \sum_{i=q+1}^{\ell} c_i$.*

**Proof** By Lemma 6.5, an LZ parse with the smallest combined length would have $c_{q+1} - c_q$ phrases of length $q$, $c_{q+2} - c_{q+1}$ phrases of length $q+1$, $c_{q+3} - c_{q+2}$ phrases of length $q+2$, etc. The longest phrases would be of length $\ell$ and their number would be $N - c_\ell + c_q - 1$. Thus a lower bound for the sum of the lengths is

$$\sum_{i=q}^{\ell-1} i\Big(c_{i+1} - c_i\Big) + \ell(N - c_\ell + c_q - 1) = (N + c_q - 1)\ell - qc_q - \sum_{i=q+1}^{\ell} c_i.$$

□

**Lemma 6.7** *The length of text $T$ is*

$$|T| \geq (N+M-1)\left(\log_c \frac{N+M-1}{M}\right) + M - \frac{N-1}{c-1}.$$

**Proof** The primary occurrences have a combined length of $Mq$. The total overlap of the $N+M$ repeats and primary occurrences is $(N+M-1)(q-1)$. Thus the length of the text is

$$\begin{aligned}|T| &= ||Z|| + Mq - (N+M-1)(q-1) \\ &\geq (N+M-1)(\ell - q + 1) - \sum_{i=q+1}^{\ell} c_i.\end{aligned}$$

The length is minimal when each $c_i$ is as large as possible. Since any substring of length $i \geq q$ consists of a $q$-gram followed by $i-q$ characters, $c_i \leq Mc^{i-q}$. Assume then that $c_i = Mc^{i-q}$ for all $i \geq q$. Since $c_\ell < (N+M-1) \leq c_{\ell+1}$, we have that $c^{\ell-q} < (N+M-1)/M \leq c^{\ell-q+1}$. Thus $\ell - q + 1 = \lceil \log_c((N+M-1)/M) \rceil = \log_c((N+M-1)/M) + b$, where $b = \lceil \log_c((N+M-1)/M) \rceil - \log_c((N+M-1)/M)$. Therefore,

$$\sum_{i=q+1}^{\ell} c_i = M \sum_{i=1}^{\ell-q} c^i = M\frac{c^{\ell-q+1} - c}{c-1} = (N+M-1)\left(\frac{c^b}{c-1}\right) - \frac{c}{c-1}M.$$

Then

$$\begin{aligned}
|T| &\geq (N+M-1)\left(\log_c \frac{N+M-1}{M} + b - \frac{c^b}{c-1}\right) + \frac{c}{c-1}M \\
&\geq (N+M-1)\left(\log_c \frac{N+M-1}{M} - \frac{1}{c-1}\right) + \frac{c}{c-1}M \\
&= (N+M-1)\left(\log_c \frac{N+M-1}{M}\right) + M - \frac{N-1}{c-1},
\end{aligned}$$

because $b - c^b/(c-1) \geq -1/(c-1)$ by Lemma 5.18. $\square$

Unlike with the LZ parse of the previous chapter, we cannot give a general upper bound on the number of phrases better than $O(|T|)$. For large enough $q$, the average length of phrases need not be much more than $q$. Then, due to the overlaps, the number of phrases is $\Theta(|T|)$. However, if we know some upper bound on $q$, or more generally, on the number $M$ of distinct $q$-grams, we can give a sublinear upper bound for $N$. This is formulated in the next theorem.

**Theorem 6.8** *If $M = O\big(|T|/(f(|T|)\log_c f(|T|))\big)$ for an increasing function $f(n)$ such that $f(n)\log_c f(n) = O(n)$, then $|Z| = O(|T|/\log_c f(|T|))$.*

**Proof** Assume that $M = O\big(|T|/(f(|T|)\log_c f(|T|))\big)$ for $f(n)$ as above. If $N = \Omega(|T|/\log_c f(|T|))$, then $\log_c((N+M-1)/M) = \Omega(\log_c f(|T|))$ and, by Lemma 6.7,

$$N \leq \frac{|T| - M + (N-1)/(c-1)}{\log_c((N+M-1)/M)} - M + 1 = O\left(\frac{|T|}{\log_c f(|T|)}\right)$$

$\square$

Applications of the theorem are given in the following corollaries.

**Corollary 6.9** *If $M = O(|T|^{1-\varepsilon})$, where $0 < \varepsilon < 1$, then we have $|Z| = O(\frac{1}{\varepsilon}|T|/\log_c |T|)$.*

**Proof** Choose $f(n) = n^{\varepsilon}/\log_c n$ in Theorem 6.8. $\square$

**Corollary 6.10** *If $q \leq (1-\varepsilon)\log_{|\Sigma|}|T|$, where $0 < \varepsilon < 1$, then we have $|Z| = O(\frac{1}{\varepsilon}|T|/\log_{|\Sigma|}|T|)$.*

**Proof** Follows from Corollary 6.9, since $M \leq |\Sigma|^q$. $\square$

## 6.3 Algorithms

Almost opposite to the previous chapter, primary searching is now easy while secondary searching is nontrivial. Since there is only one primary occurrence for each $q$-gram, the primary search is executed as an access query for the set $\mathcal{Q}$ in the universe $\Sigma^q$. It can be implemented using, e.g., hashing in a constant time, assuming, as we do, that a $q$-gram fits into constant number of words. The space requirement is $O(|\mathcal{Q}| \log |T|)$ bits. Note that, unlike in the previous chapter, the text itself is not needed. This part of the query is identical with the traditional $q$-gram indexes.

### 6.3.1 Secondary search

The remaining part of the query, finding all the secondary occurrences, is again solved using the general secondary search method described in Algorithm 1 on page 57, where the idea is to find, for each occurrence of the query $q$-gram, the sources that contain it. Recall that Algorithm 1 has two non-trivial implementation details, the range containment query on line 6 and the check for duplicate occurrences on line 8. Since the repeats of an LZ index for $q$-grams cannot overlap by more than $q-1$ positions, there can be no duplicate occurrences, and the check on line 8 is not needed.

For the range containment problem, we can use the same solution as with the regular LZ parses in Section 5.2.3: linearize the phrase set (Lemma 5.14) and store the starting points and ending points of sources in bucketed bitmaps (Theorem 3.1). The space requirement is $O(|T| + |Z| \log |T|)$ bits and the query time $O(1 + k)$ for $k$ occurrences. We have the following result.

**Theorem 6.11** *Let $T$ be a text with $M$ distinct $q$-grams and $Z$ its LZ parse for $q$-grams. Then $T$ and $Z$ can be stored in $O(|T| + (|Z| + M) \log |T|)$ bits so that all occurrences of any $q$-gram can be found in time $O(1 + k)$, where $k$ is the number of occurrences.*

The worst case lower bound for $|Z|$ is $\Omega(|T|/\log_{|\Sigma|} |T|)$, independent of $q$, and we need $\Omega(|T| \log |\Sigma|)$ bits just for storing the phrases. Thus the bucketed bitmap method is, in this case, an optimal solution for the range containment problem, both in time and space. However, if the text is compressible, it is possible that the space requirement of the phrases, $O(|Z| \log |T|)$ bits, is sublinear in $|T|$. Then the space requirement of the bucketed bitmaps, $\Omega(|T|)$ bits, dominates. Note also that the space requirement of primary searching, $O(|\mathcal{Q}| \log |T|)$ bits, can also be $o(|T|)$. In Section 6.4, we describe a new data structure, called the nesting leveled list, for the range containment problem. It has the space requirement of

$O(|Z| \log |T|)$ bits and the query time of $O(1+k)$. Thus, it is optimal also in the case of compressible text. We have the following improved result.

**Theorem 6.12** *Let $T$ be a text with $M$ distinct $q$-grams and $Z$ its LZ parse for $q$-grams. Then $T$ and $Z$ can be stored in $O((|Z|+M)\log |T|)$ bits so that all occurrences of any $q$-gram can be found in time $O(1+k)$, where $k$ is the number of occurrences.*

### 6.3.2 Comparison to traditional $q$-gram indexes

The traditional $q$-gram index has the same time requirement as the LZ index, but the space requirement is $O(|T| \log |T|)$. The space advantage of the LZ index depends on the number $M$ of distinct $q$-grams. For large $M$, $|Z| + M$ can be $\Theta(|T|)$ and there is no space advantage. On the other hand, Corollaries 6.9 and 6.10 give us the following cases, where the space requirement is reduced by the factor $\Omega(\log_{|\Sigma|} |T|)$.

**Theorem 6.13** *If $M = O(|T|^{1-\varepsilon})$ for a constant $0 < \varepsilon < 1$, then $T$ can be stored in $O(\frac{1}{\varepsilon}|T| \log |\Sigma|)$ bits so that all occurrences of any $q$-gram can be found in time $O(1+k)$.*

**Theorem 6.14** *If $q \leq (1-\varepsilon)\log_{|\Sigma|} |T|$ for a constant $0 < \varepsilon < 1$, then $T$ can be stored in $O(\frac{1}{\varepsilon}|T| \log |\Sigma|)$ bits so that all occurrences of any $q$-gram can be found in time $O(1+k)$.*

In the the approximate string matching methods using a $q$-gram index in [JU91, HS94, NBY98], the optimal value of $q$ depends on the pattern $(P, d)$ and the type of the text (e.g., the alphabet size), but not on the length of the text. Then, for these methods, the LZ index has an asymptotically smaller space requirement than traditional $q$-gram indexes provided that $\log |\Sigma| = o(\log |T|)$.

In the $q$-gram method by Myers, the optimal value of $q$, for a random text, is $\log_{|\Sigma|} |T|$ [Mye94]. For this value, the LZ index offers no space saving. However, it is possible to use a smaller value of $q$, but at the cost of increased query time. The precise effect of $q$ is difficult to analyze, and the analysis is outside of the scope of this dissertation, but based on the analysis in [Mye94], if $t(q)$ is the expected query time for the value $q$ in a random text, the slowdown factor $t((1-\varepsilon)q)/t(q)$ is roughly $(|T|/t(q))^{\varepsilon}$. Anyway, the LZ index offers the possibility for a space–time tradeoff.

## 6.4 Nesting leveled list

Next, we present a new data structure for the range containment problem that rises in secondary searching (line 6 in Algorithm 1). The core data structure, called the *nesting leveled list* is a general data structure for the range containment problem, but it leaves one aspect of the solution open. With additional data structures this open aspect is solved efficiently for the special case of secondary searching with an LZ parse for $q$-grams.

### 6.4.1 Nesting levels

In Section 2.2.2, we described two orthogonal partial orders for ranges: containment and precedence. Containment with distinct end points was called nesting. The nesting relation plays a key role in our data structure.

**Definition 6.15** Let $R$ be a set of ranges. The *nesting depth* of a range $[i, j]$ in $R$, denoted by $nl_R([i, j])$, is the largest integer $\ell$ for which there are ranges $\{[i_1, j_1], \ldots, [i_\ell, j_\ell]\}$ in $R$ such that $[i, j] \Subset [i_1, j_1] \Subset \cdots \Subset [i_\ell, j_\ell]$.

**Definition 6.16** The $\ell$th *nesting level* of $R$, denoted by $NL_\ell(R)$ is the set of ranges in $R$ with the nesting depth $\ell$, i.e., $NL_\ell(R) = \{[i, j] \in R \mid nl_R([i, j]) = \ell\}$. Let $H$ be the largest integer for which $NL_H(R)$ is nonempty. Then, $NL_0(R), NL_1(R), \ldots, NL_H(R)$ is the *nesting level partition* of $R$.

The importance of nesting levels is based on the following results.

**Lemma 6.17** *Two intervals $[i, j]$ and $[i', j']$ are incomparable in $\leq$ iff $[i, j] \Subset [i', j']$ or $[i', j'] \Subset [i, j]$.*

**Proof** Follows from Lemma 2.11. $\square$

**Theorem 6.18** *Let $R$ be a set of intervals and $\ell$ a non-negative integer. Then, $\leq$ is a total order for $NL_\ell(R)$.*

**Proof** Let $[i, j]$ and $[i', j']$ be intervals in $NL_\ell(R)$. Assume, contradictory to the theorem, that $[i, j]$ and $[i', j']$ are incomparable in $\leq$. Then, by Lemma 6.17, $[i, j] \Subset [i', j']$ or $[i', j'] \Subset [i, j]$. Assume $[i, j] \Subset [i', j']$. Then $nl_R([i, j]) > nl_R([i', j'])$ which is a contradiction. Thus $[i, j]$ and $[i', j']$ are comparable in $\leq$. $\square$

$$
\begin{tikzcd}
{[1,3]} \arrow[r, leftrightarrow] & {[1,11]} \arrow[r, leftrightarrow] & {[15,24]} \\
 & {[2,7]} \arrow[u] \arrow[r, leftrightarrow] & {[19,21]} \arrow[u]
\end{tikzcd}
$$

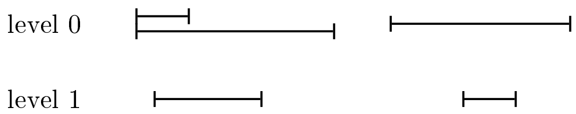



Figure 6.2: Two representations of the nesting leveled list for the LZ parse of Figure 6.1.

### 6.4.2 Nesting leveled list

Now we are ready to describe the new data structure *nesting leveled list* (NLL). The range containment problem it solves is to find the ranges that contain a query range. In our case, the set of ranges is the set $\mathcal{I}$ of sources of an LZ parse for $q$-grams, and the query range is an occurrence $O$ of a $q$-gram $Q$. In this subsection, we will not take advantage of any special properties of this special case.

The nesting leveled list is based on the concepts of the previous section. Let $NL_0(\mathcal{I}), NL_1(\mathcal{I}), \ldots, NL_H(\mathcal{I})$ be the nesting level partition of the set $\mathcal{I}$. The nesting leveled list of $\mathcal{I}$, NLL($\mathcal{I}$), consists of $H+1$ $\leq$-ordered bidirectional lists, one for each nesting level $NL_\ell(\mathcal{I})$, $\ell = 0, \ldots, H$. Recall that $\leq$ is a total order for $NL_\ell(\mathcal{I})$. There are also links between the nesting levels, called *parent links*, from every source to one of its parents. A *parent* of a source $I \in NL_\ell(\mathcal{I})$ is a source $I' \in NL_{\ell-1}(\mathcal{I})$ such that $I \Subset I'$. Every source $I$ has one or more parents unless $I \in NL_0(\mathcal{I})$. The operations Pred, Succ and Parent supported by the NLL are defined as follows.

**Definition 6.19** Let $I$ be the immediate predecessor of $I'$ in the list for their common nesting level. Then $\textsc{Pred}(I') = I$ and $\textsc{Succ}(I) = I'$. If $I$ does not have a predecessor (successor), then $\textsc{Pred}(I) = \textsc{null}$ ($\textsc{Succ}(I) = \textsc{null}$).

**Definition 6.20** Let $I \in NL_\ell(\mathcal{I})$, $\ell > 0$. Then $\textsc{Parent}(I) = I'$ for some $I'$ such that $I' \in NL_{\ell-1}(\mathcal{I})$ and $I \Subset I'$. If $I \in NL_0(\mathcal{I})$, then $\textsc{Parent}(I) = \textsc{null}$.

Examples of NLLs are shown in Figures 6.2 and 6.3.

Suppose we know that a source $I \in NL_\ell(\mathcal{I})$ contains an occurrence $O$ of the $q$-gram $Q$. Then the sources in $NL_\ell(\mathcal{I})$ containing $O$ form a continuous neighborhood of $I$ in the list for $NL_\ell(\mathcal{I})$ and they can be found quickly using the operations Pred and Succ. That is, one just needs to follow the list in both directions until encountering a source that does not contain $O$.

Furthermore, every parent of $I$ (particularly PARENT($I$)) also contains $O$. Therefore, starting from $I$, using PARENT to change level and using PRED and SUCC to search the list on each level, we can find all sources in the nesting levels $0, \dots, \ell$ that contain $O$. The method is described in detail in Procedure EXTENDFIND.

**Procedure** EXTENDFIND
**Input:** occurrence $O$ of a $q$-gram, source $I$ that contains $O$
**Output:** all sources containing $O$ on the nesting level of $I$ and on the higher nesting levels

```
S := ∅
repeat
    S := S ∪ {I}
    I' := PRED(I)
    while I' ≠ NULL and O ⊆ I' do          % search left
        S := S ∪ {I'}
        I' := PRED(I')
    I' := SUCC(I)
    while I' ≠ NULL and O ⊆ I' do          % search right
        S := S ∪ {I'}
        I' := SUCC(I')
    I := PARENT(I)                % move to next level
until I = NULL
return S
```

Each execution of an inner loop body in EXTENDFIND adds a new source containing $O$ into $\mathcal{S}$. Thus EXTENDFIND takes $O(|\mathcal{S}|)$ time. The space requirement of the data structures is $O(|Z| \log |T|)$ bits.

Given Procedure EXTENDFIND, the range containment problem is easy to solve if we can find a proper starting point, i.e., a source that contains $O$ in the deepest nesting level where there is one. Finding the starting point efficiently requires additional knowledge about the occurrence $O$ and the set $\mathcal{I}$. The additional knowledge we use depends on whether $O$ is a primary or a secondary occurrence.

### 6.4.3 Starting point for primary occurrences

For primary occurrences, the starting points are found using precomputed START-values.

**Definition 6.21** Let $O$ be the primary occurrence of a $q$-gram. Then, START($O$) $= I$ for some $I$ such that $O \subseteq I \in NL_\ell(\mathcal{I})$, where $\ell$ is the deepest level that has sources containing $O$. If no source contains $O$, then $\textsc{Start}(O) = \textsc{null}$.

The following algorithm now solves the range containment problem.

**Algorithm 5** Range containment for primary occurrences
**Input:** Primary occurrence $O$ of a $q$-gram
**Output:** All sources containing $O$

$I := \textsc{Start}(O)$
**if** $I = \textsc{null}$ **then return** $\emptyset$
**else return** $\textsc{ExtendFind}(I, O)$

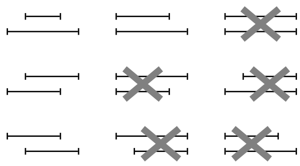
Algorithm 5 uses $O(1+k)$ time to report $k$ sources. The $\textsc{Start}$ pointers require $O(M \log |Z|)$ space for $M$ distinct $q$-grams.

### 6.4.4 Starting point for secondary occurrences

We cannot afford to store a precomputed starting point for every secondary occurrence. Instead, we will make use of two pieces of additional knowledge about the occurrence $O$ and the set $\mathcal{I}$ of sources. First, because $O$ is a secondary occurrence, we know a repeat $J$ that contains $O$. Second, by Lemma 6.1, the repeat $J$ cannot contain any sources.

The potential starting points are the sources that overlap the repeat $J$ by at least $q$ positions. As $J$ cannot completely contain a source, we are left with the following possibilities of how such a source $I$ lies with respect to $J$.

1. Containing source: $J \subset I$
2. Preceding source: $I \ll J$
3. Succeeding source: $J \ll I$



The first category sources are guaranteed to contain the occurrence $O$. The preceding and succeeding sources may or may not contain $O$.

Let $K_J$ be the deepest nesting level that has a source containing $J$. If there are no containing sources, define $K_J = -1$. Let $H_J$ be the deepest nesting level that has a containing, preceding or succeeding source with an overlap of at least $q$. Again, if there are no such sources, define $H_J = -1$. Now all nesting levels $0, \ldots, K_J$ have a source that contains $O \subseteq J$, and the starting point for $\textsc{ExtendFind}$ must be between levels $K_J$ and $H_J$, inclusive.

The *preceding edge* for a repeat $J$ contains the last preceding source from each of the levels $K_J + 1, \ldots, H_J$. Similarly, the *succeeding edge* contains

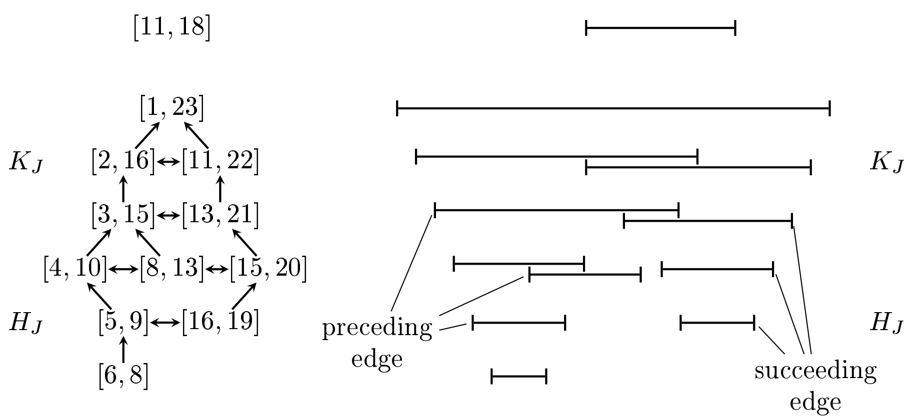



Figure 6.3: The figure shows two representations of the same nesting leveled list. At the top is the repeat $J$ containing the current 3-gram occurrence. The levels $K_J$ and $H_J$, and the corresponding preceding and succeeding edges are indicated in the figure.

the first succeeding source from each of the levels $K_J + 1, \ldots, H_J$. As the levels $K_J + 1, \ldots, H_J$ have no containing sources, the last preceding source and the first succeeding source on each level are adjacent on the list of their nesting level. If neither of the two contains $O$, then neither does any other source on the same level. Therefore, it is enough to search the preceding and succeeding edges for the starting point for EXTENDFIND. Preceding and succeeding edges are illustrated in Figure 6.3.

The range containment query for a secondary occurrence is described in detail in Algorithm 6. The algorithm uses BOTTOM and PRECEDGE which are precomputed values defined as follows.

**Definition 6.22** Let $J$ be a repeat and let $K_J$ and $H_J$ be as above. If $K_J = -1$, $\text{BOTTOM}(J) = \text{NULL}$. Otherwise $\text{BOTTOM}(J) = I$ for some $I$ that is in $NL_{K_J}(\mathcal{I})$ and contains $J$. For $1 \le \ell \le H_J - K_J$, $\text{PRECEDGE}(J, \ell) = I$ such that $I$ is the last source in $NL_{K_J+\ell}(\mathcal{I})$ that precedes $J$. Additionally, $\text{PRECEDGE}(J, H_J - K_J + 1) = \text{NULL}$.

If necessary, we will add a dummy source (e.g. $[1, 0]$) at the start of every nesting level list so that $\text{PRECEDGE}(J, \ell)$ is defined for all $J$ and all $\ell = 1, \ldots, H_J - K_J$.

**Algorithm 6** Range containment for secondary occurrence
**Input:** secondary occurrence $O$ of a $q$-gram, repeat $J$ containing $O$
**Output:** all sources containing $O$

$I := \textsc{Bottom}(J)$
$\ell := 1$
$I' := \textsc{PrecEdge}(J, 1)$
**while** $I' \neq$ NULL **and** $(O \subseteq I'$ **or**
  $\textsc{Succ}(I') \neq$ NULL **and** $O \subseteq \textsc{Succ}(I'))$ **do**
  $I := I'$
  $\ell := \ell + 1$
  $I' := \textsc{PrecEdge}(J, \ell)$
**if** $I =$ NULL **then return** $\emptyset$
**else**
  **if** $O \not\subseteq I$ **then** $I := \textsc{Succ}(I)$
  **return** $\textsc{ExtendFind}(I, O)$

Each round of the **while** loop in the algorithm passes a source that contains $O$ (either $I'$ or $\textsc{Succ}(I')$). Each of those sources will be found and returned by ExtendFind. Thus the algorithm uses $O(1 + k)$ time to find and report $k$ sources. The space requirement for the Bottom pointers is clearly $O(|Z| \log |T|)$ bits. For each value $\textsc{PrecEdge}(J, \ell)$, $1 \leq \ell \leq H_J - K_J$, there is at least one preceding or succeeding source that overlaps $J$ by at least $q$ positions. Repeats cannot overlap each other by more than $q - 1$ positions. Thus, a source may overlap at most two repeats by $q$ positions without containing them. Therefore, the total number of elements on the PrecEdge lists is at most $2|Z|$, and the total space requirement is $O(|Z| \log |T|)$ bits.

### 6.4.5 Construction

We will next sketch an algorithm that constructs the nesting leveled list, and the augmenting data structures, the Start, Bottom and PrecEdge pointers, given the output of Algorithm 4.

Algorithm 4 outputs three kinds of intervals: sources, repeats and primary occurrences. The construction of NLL starts by sorting all these intervals together by the starting position. The intervals are then processed in the sorted order:

- When a source is encountered, it is added to the (originally empty) nesting leveled list NLL.

- When a repeat $J$ is encountered, $\textsc{Bottom}(J)$ and $\textsc{PrecEdge}(J, \ell)$, for $1 \leq \ell \leq H_J - K_J$, are computed.

- When a primary occurrence $O$ is encountered, the value $\textsc{Start}(O)$ is computed.

A more detailed description of the construction algorithm is given below.

1. All three kinds of intervals are sorted together into increasing order of the starting positions. Among intervals with the same starting position, sources are placed before repeats and occurrences, and the end position is used as an additional key if necessary.

2. The intervals are processed in the sorted order. Let $[i, j]$ be the interval to be processed next and let $K$ be the deepest level in the current NLL that has a source $I'$ containing $[i, j]$. The level $K$ and the source $I'$ can be found, e.g., by a binary search among the latest sources of the nesting levels.

3. The rest of the processing of the interval $[i, j]$ depends on its type.

   - Let $[i, j]$ be the source $I$. If $I \Subset I'$, then add $I$ into NLL as the last source on level $K + 1$ and set $\textsc{Parent}(I) = I'$. Otherwise (i.e., when $\overrightarrow{I} = \overrightarrow{I'}$) add $I$ into NLL as the last source on level $K$ and set $\textsc{Parent}(I) = \textsc{Parent}(I')$.

     In the former case, $I$ may belong to the succeeding edge of the latest repeat $J'$. Following Section 6.4.4, let $K_{J'}$ and $H_{J'}$ be such that the current $\textsc{PrecEdge}$ list for $J'$ is from level $K_{J'} + 1$ to $H_{J'}$. If $H_{J'} = K$ and $I$ overlaps $J'$ by at least $q$ characters, then set $\textsc{PrecEdge}(J', K + 1 - K_{J'}) = \textsc{Prec}(I)$. If $\textsc{Prec}(I) = \textsc{null}$ (i.e., $I$ is the first source on level $K + 1$), add a dummy source at the start of level $K + 1$.

   - If $[i, j]$ is the repeat $J$, set $\textsc{Bottom}(J) = I'$. Additionally, for $\ell = 1, 2, \ldots$, set $\textsc{PrecEdge}(J, \ell) = I_\ell$, where $I_\ell$ is the currently last source on level $K + \ell$, until $I_\ell$ no more overlaps $J$ by at least $q$ positions.

   - If $[i, j]$ is the occurrence $O$, set $\textsc{Start}(O) = I'$.

The construction requires $O((|Z| + |\mathcal{Q}|) \log(|Z| + |\mathcal{Q}|))$ time and $O((|Z| + |\mathcal{Q}|) \log |T|)$ space. The time can be reduced by using, e.g., bucket sorting.

## 6.5 Concluding remarks

The remarkable feature of the LZ index for $q$-grams is that it is asymptotically optimal both in time and space when $q$ is a constant or grows slowly enough in the length of the text. No algorithm can find and report $k$ occurrences in better than $\Theta(1+k)$ time, and unless the text is compressible, no index can store the information about all $q$-grams of the text in less than $\Theta(|T| \log |\Sigma|)$ bits. Furthermore, if the text does compress well, then so does the LZ index for $q$-grams, too.

# Chapter 7

# Concluding remarks

Repetition-based indexing is a new way of looking at text indexing problems. Its main idea is to reduce the work done in pattern matching by using precomputed information about repetitions in the text. All text indexes are based on the same principle but often implicitly. With repetition-based indexes, repetition information is the center piece around which the data structures and algorithms are build. This information-centric approach, focusing at first on the relevant information rather than on particular data structures and algorithms, leads to a more general view of the problem.

The repetition-based indexing scheme we described provides a general framework for text indexing based on representing repetition information by phrases. We have seen that starting with different phrase sets and developing algorithms and data structures for them in the framework can lead to previously known text indexes as well as to new improved indexes.

We have also discovered subtle connections between indexed string matching and some other problems. The connection to one-dimensional range searching is quite obvious, but the central role of two-dimensional range searching in the Lempel–Ziv index is rather surprising considering the one-dimensional nature of strings. The common factor of the Lempel–Ziv index and Ziv–Lempel text compression is the Lempel–Ziv parse. The relevant property of Lempel–Ziv parses for both applications is the extensive but *nonredundant* repetition information.

There are many possibilities for further development of the methods.

- Better algorithms and data structures for the subproblems could lead to improvements of the present results. In particular, the primary searching phase of the Lempel–Ziv index may still have room for improvement.

- Different phrase sets can be used. Some ideas were mentioned in

Section 5.3, including non-1-spaced phrase sets and phrase sets based on word-based Huffman coding [BSTW86, Mof89].

- The scheme itself might be extended beyond the formalization given here. For example, a source might be associated with multiple repeats or even with approximate repeats.

We have only considered string matching and its special case, $q$-gram matching. The scheme may also be used with other types of patterns. In particular, the secondary searching phase is not sensitive to the type of the pattern. For primary searching, new methods may have to be developed for different types of patterns.

We have concentrated on asymptotic results. Many of the methods are not practical. However, with simpler data structures, such as the $k$-d tree [Ben75] suggested for two-dimensional range searching in [KU96a], practical indexes may be possible. The $q$-gram index, especially the version using bucketed bitmaps (Theorem 6.11), might be useful in the presented form. Preliminary tests indicate that in both cases it is possible to get below the space requirement of traditional indexes also in practice. Some preliminary estimates of the practical size of the (NLL-based) $q$-gram index were given in [KS98]. There are plans for further experimentation.

Viewing problems from different directions often leads to better understanding of the problems and the difficulties involved. Repetition-based indexing can provide a new and quite different viewpoint to many pattern matching problems. The information-centric approach could also be useful for many other problems.

# References


[ABF96] Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, April 1996.

[Aga97] Pankaj K. Agarwal. Range searching. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–598. CRC Press, 1997.

[AMRT96] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on $AC^0$ RAMs: Query time $\theta(\sqrt{\log n / \log\log n})$ is necessary and sufficient. In *Proc. 37th IEEE Annual Symposium on Foundations of Computer Science*, pages 441–450. IEEE Computer Society Press, 1996.

[AMT99] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. Fusion trees can be implemented with $AC^0$ instructions only (Note). *Theoretical Computer Science*, 215:337–344, February 1999.

[AN93] Arne Andersson and Stefan Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, July 1993.

[AN95] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Software — Practice and Experience*, 25(2):129–141, February 1995.

[And96] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th IEEE Annual Symposium on Foundations of Computer Science*, pages 135–141. IEEE Computer Society Press, 1996.

[BBH+85] Anselm Blumer, Janet A. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40(1):31–55, September 1985.

[BBH+87] Anselm Blumer, Janet A. Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[BCW90] Timothy Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[BEH89] Anselm Blumer, Andrzej Ehrenfeucht, and David Haussler. Average sizes of suffix trees and DAWGs. *Discrete Applied Mathematics*, 24(1–3):37–45, August 1989.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Ben80] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, April 1980.

[BSTW86] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[BWC89] Timothy Bell, Ian H. Witten, and John G. Cleary. Modelling for text compression. *ACM Computing Surveys*, 21(4):557–592, December 1989.

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

[CD96] Livio Colussi and Alessia De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, June 1996.

[Cha88] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.

[CHM97] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1–2):1–143, August 1997.

[CL94] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 15(4/5):327–344, October/November 1994.

[Cob95] Archie L. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1995.

[Cro86] Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63–86, 1986.

[CV97] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graph. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1997.

[DFG+97] Gautam Das, Rudolf Fleischer, Leszek Gąsieniec, Dimitris Gunopulos, and Juha Kärkkäinen. Episode matching. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 1997.

[dMNZ97] Edleno Silva de Moura, Gonzalo Navarro, and Nivio Ziviani. Indexing compressed text. In *Proc. 4th South American Workshop on String Processing*, pages 95–111. Carleton University Press, 1997.

[FG89] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, April 1989.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[FT98] Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, April 1998.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

[Gal82] John Kenneth Gallant. *String Compression Algorithms.* PhD Thesis, Department of Electrical Engineering and Computer Science, Princeton University, June 1982.

[GBY91] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of algorithms and data structures in Pascal and C.* Addison–Wesley, 1991.

[GBYS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 5, pages 66–82. Prentice–Hall, 1992.

[GKPR96] Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel–Ziv encoding. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 1996.

[GKS99] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 1999.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

[HS94] Niklas Holsti and Erkki Sutinen. Approximate string matching using $q$-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. Department of Computer Science, University of Joensuu, 1994.

[HT84] Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[Irv95] Rob Irving. Suffix binary search trees. Technical Report TR-1995-7, Department of Computing Science, University of Glasgow, April 1995.

[JU91] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th International Symposium on Mathematical Foundations of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer, 1991.

[Kär95] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 1995.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison–Wesley, second edition, 1998.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 869–941. Elsevier, 1990.

[KS98] Juha Kärkkäinen and Erkki Sutinen. Lempel–Ziv index for $q$-grams. *Algorithmica*, 21(1):137–154, May 1998.

[KTS+98] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, Masamichi Miyazaki, and Setsuo Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference*, pages 103–112. IEEE Computer Society Press, 1998.

[KTSA99] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Shift-and approach to pattern matching in LZW compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching*, volume 1645 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1999.

[KU96a] Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing*, pages 141–155. Carleton University Press, 1996.

[KU96b] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and*

*Combinatorics*, volume 1090 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 1996.

[Kur98] Stefan Kurtz. Reducing the space requirement of suffix trees. Report 98-03, Technische Fakultät, Universität Bielefeld, 1998.

[LZ76] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(1):75–81, January 1976.

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

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

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

[Mof89] Alistair Moffat. Word-based text compression. *Software — Practice and Experience*, 19(2):185–198, February 1989.

[Mor68] Donald R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[Mye94] Eugene W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, October/November 1994.

[Mye98] Eugene W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1998.

[Nav98] Gonzalo Navarro. *Approximate Text Searching*. PhD Thesis, Department of Computer Science, University of Chile, 1998.

[NBY98] Gonzalo Navarro and Ricardo Baeza-Yates. A practical $q$-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. [`http://www.clei.cl`].

[NR99] Gonzalo Navarro and Mathieu Raffinot. A general practical approach to pattern matching over Ziv–Lempel compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching*, volume 1645 of *Lecture Notes in Computer Science*, pages 14–36. Springer, 1999.

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

[Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proc. 4th Annual European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 1996.

[ST96] Erkki Sutinen and Jorma Tarhio. Filtration with $q$-samples in approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 50–63. Springer, 1996.

[Sto88] James A. Storer. *Data Compression: Methods and Theory.* ACM Press, 1988.

[Sut98] Erkki Sutinen. *Approximate Pattern Matching with the q-Gram Family.* PhD Thesis, Report A-1998-3, Department of Computer Science, University of Helsinki, 1998.

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

[Ukk93] Esko Ukkonen. Approximate string-matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1993.

[Ukk95] Esko Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, September 1995.

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

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, August 1983.

[Wil84] Dan E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, June 1984.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

ISSN 1238-8645
ISBN 951-45-8917-3
Helsinki 1999
Helsinki University Printing House

**Juha Kärkkäinen: Repetition-Based Text Indexes** **A-1999-4**