Data Structures for Efficient String Algorithms

Johannes Fischer

München 2007

Data Structures for Efficient String Algorithms

Johannes Fischer

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Johannes Fischer
aus Kronberg i. Ts.

München, den 10. Oktober 2007

Erstgutachter: Prof. Dr. Volker Heun
Zweitgutachter: Prof. Dr. Enno Ohlebusch
Tag der mündlichen Prüfung: 8. Oktober 2007

Abstract

This thesis deals with data structures that are mostly useful in the area of string matching and string mining. Our main result is an $O(n)$-time preprocessing scheme for an array of n numbers such that subsequent queries asking for the position of a minimum element in a specified interval can be answered in constant time (so-called RMQs for *Range Minimum Queries*). The space for this data structure is $2n + o(n)$ bits, which is shown to be asymptotically optimal in a general setting. This improves all previous results on this problem. The main techniques for deriving this result rely on combinatorial properties of arrays and so-called Cartesian Trees. For compressible input arrays we show that further space can be saved, while not affecting the time bounds. For the two-dimensional variant of the RMQ-problem we give a preprocessing scheme with quasi-optimal time bounds, but with an asymptotic increase in space consumption of a factor of $\log n$.

It is well known that algorithms for answering RMQs in constant time are useful for many different algorithmic tasks (e.g., the computation of lowest common ancestors in trees); in the second part of this thesis we give several new applications of the RMQ-problem. We show that our preprocessing scheme for RMQ (and a variant thereof) leads to improvements in the space- and time-consumption of the *Enhanced Suffix Array*, a collection of arrays that can be used for many tasks in pattern matching. In particular, we will see that in conjunction with the suffix- and LCP-array $2n + o(n)$ bits of additional space (coming from our RMQ-scheme) are sufficient to find all *occ* occurrences of a (usually short) pattern of length m in a (usually long) text of length n in $O(m\sigma + occ)$ time, where σ denotes the size of the alphabet. This is certainly optimal if the size of the alphabet is constant; for non-constant alphabets we can improve this to $O(m \log \sigma + occ)$ locating time, replacing our original scheme with a data structure of size $\approx 2.54\, n$ bits. Again by using RMQs, we then show how to solve frequency-related string mining tasks in optimal time. In a final chapter we propose a space- and time-optimal algorithm for computing suffix arrays on texts that are logically divided into words, if one is just interested in finding all word-aligned occurrences of a pattern.

Apart from the theoretical improvements made in this thesis, most of our algorithms are also of practical value; we underline this fact by empirical tests and comparisons on real-word problem instances. In most cases our algorithms outperform previous approaches by all means.

Zusammenfassung

Diese Arbeit beschäftigt sich mit Datenstrukturen, die insbesondere im String Matching und String Mining Anwendung finden. Das Hauptresultat ist ein $O(n)$-Konstruktionsalgorithmus für eine Datenstruktur der Größe $2n + o(n)$ Bits, die es erlaubt, für ein Feld von n Zahlen die Position eines minimalen Elements in einem beliebigen Anfrageintervall in konstanter Zeit zu finden (sog. RMQs für *Range Minimum Queries*). Dies verbessert alle vorherigen Resultate zu diesem Problem. Die Haupttechniken, die zu diesem Ergebnis führen, beruhen auf kombinatorischen Eigenschaften von Feldern und sog. Kartesischen Bäumen. Wir zeigen zudem, dass dieses Schema zur Präprozessierung eines Feldes für konstante RMQs im allgemeinen Fall asymptotisch optimal ist, man jedoch für komprimierbare Eingabefelder weiteren Platz einsparen kann, ohne dass die Zeitoptimalität leidet. Für die zweidimensionale Variante desselben Problems stellen wir einen quasi-zeitoptimalen Algorithmus vor, dessen Platzbedarf jedoch etwas höher ist (asymptotisch um den Faktor $\log n$).

Neben den bereits bekannten Algorithmen, die RMQs zur Lösung bestimmter Probleme verwenden (z.B. die Berechnung niedrigster gemeinsamer Vorfahren in Bäumen), betrachten wir im zweiten Teil dieser Dissertation neue Anwendungen des RMQ-Problems. Wir zeigen, dass die oben genannte Datenstruktur (und eine Variante hiervon) zu Verbesserungen im Zeit- und Platzbedarf des *Enhanced Suffix Array* führt, einer Sammlung von Feldern zur Lösung vieler String-Matching-Probleme. Insbesondere werden wir sehen, dass die $2n + o(n)$ Bits von unserer RMQ-Struktur an zusätzlichem Platz ausreichen, um mit dem Suffix- und LCP-Array alle vk Vorkommen eines (normalerweise kleinen) Suchtextes der Länge m in einem (normalerweise großen) Text der Länge n in Zeit $O(m\sigma + vk)$ zu finden, wobei σ die Größe des Alphabets bezeichnet. Für konstante Alphabetgrößen ist dies sicherlich optimal; im Falle nicht konstanter Alphabetgrößen können wir dieses Resultat jedoch verbessern: ca. $2.54\,n + o(n)$ zusätzliche Bits reichen aus, um mit dem Suffix- und LCP-Array alle vk Vorkommen eines Suchtextes in Zeit $O(m \log \sigma + vk)$ zu finden. Wiederum durch Anwendung unseres Resultats über RMQs zeigen wir dann, wie wir häufigkeitsbasierte String-Mining-Anfragen in Textdatenbanken in optimaler Zeit lösen können. In einem abschließenden Kapitel geben wir einen zeit- und platzoptimalen Konstruktionsalgorithmus für Suffix Arrays auf Texten, die in logische Einheiten (z.B. natürlichsprachige Wörter) eingeteilt sind, wenn man bei der Mustersuche nur an Treffern interessiert ist, die an Wortgrenzen beginnen.

Die meisten der in dieser Dissertation vorgestellten Resultate haben neben ihrer theoretischen Relevanz auch praktische Bedeutung; eine Tatsache, die mittels empirischer Tests und Vergleichen auf realen Daten unterstrichen wird. In den meisten Fällen verbessern unsere neuen Algorithmen die Laufzeiten und den Platzbedarf früherer Ansätze deutlich.

CONTENTS

1 Introduction **1**
1.1 Synopsis 2
1.2 Practicability of the Results 5
1.3 Previous Publications 5
1.4 A Note on Citation Policy 6

2 Basic Concepts **7**
2.1 Arrays and Strings 7
2.2 Basic String Matching Problems 8
2.3 Suffix Trees 9
2.4 Suffix- and LCP-Arrays 10
2.5 Text Compressibility and Empirical Entropy 11
2.6 Compact Data Structures for Rank and Select 12
2.7 Succinct Representations of Suffix- and LCP-Arrays 13
2.8 Balanced Parentheses Representation of Trees 14

3 An Optimal Preprocessing Scheme for RMQ **17**
3.1 Chapter Introduction 17
3.2 Preliminary Definitions 18
3.3 Applications of RMQ 19
3.3.1 Computing Lowest Common Ancestors in Trees 19
3.3.2 Computing Longest Common Extensions of Suffixes . . . 20
3.3.3 (Re-)Construction of Suffix Links 21
3.3.4 Document Retrieval Queries 22
3.3.5 Maximum-Sum Segment Queries 22
3.3.6 Lempel-Ziv-77 Data Compression 22
3.4 Previous Results on RMQs 23
3.4.1 The Berkman-Vishkin Algorithm 23

3.4.2 Alstrup et al.'s Idea for Handling In-Block-Queries 25
3.4.3 Sadakane's Succinct RMQ-Algorithm 26
3.5 An Improved Algorithm 29
3.5.1 A New Code for Binary Trees 30
3.5.2 The Algorithm 36
3.6 A Lower Bound 41
3.7 Applications of the New Algorithm 42
3.7.1 LCAs in Trees with Small Average Degree 42
3.7.2 An Improved Algorithm for Longest Common Extensions 43
3.8 Practical Considerations 44
3.9 How to Further Reduce Space 51
3.9.1 Small "Alphabets" Don't Help! 51
3.9.2 Compression Techniques 52
3.9.3 Outlook 55
3.10 Summary and Discussion 56

4 2-Dimensional RMQs 57
4.1 Chapter Introduction 57
4.2 Preliminaries 58
4.3 Methods 58
4.3.1 A General Trick for Query Precomputation 59
4.3.2 Linear Preprocessing of the First Level 60
4.3.3 Recursive Partitioning 62
4.3.4 What's Left: How to Find the Right Grid 64
4.4 Summary and Outlook 65

5 Improvements in the Enhanced Suffix Array 67
5.1 Chapter Introduction 67
5.2 Enhanced Suffix Arrays 68
5.3 An RMQ-based Representation of the Child-Table 70
5.4 Pattern Matching in $O(m|\Sigma|)$ Time 71
5.5 Pattern Matching in $O(m \log |\Sigma|)$ Time 72
5.5.1 Basic Idea 72
5.5.2 A Pseudo-Median Algorithm for RMQ on LCP 73
5.5.3 Summing Up 81

6 String Mining Problems 83
6.1 Chapter Introduction 83
6.2 Formal Problem Definition 84
6.3 The New Algorithm 86
6.3.1 Preprocessing 88
6.3.2 Labeling 89
6.3.3 Extraction 90
6.3.4 Reducing the Size of the Output 93
6.4 Practical Performance 94
6.5 Conclusions 98

7 Suffix Arrays on Words **99**
7.1 Chapter Introduction 99
7.2 Chapter Outline 101
7.3 Definitions 102
7.4 Optimal Construction of the Word Suffix Array 102
7.5 Word-Based LCP-Arrays 105
7.6 Searching in the Word Suffix Array 107
7.6.1 Searching in $O(m \log k)$ Time 107
7.6.2 Searching in $O(m + \log k)$ Time 109
7.6.3 Searching in $O(m|\Sigma|)$ and $O(m \log |\Sigma|)$ Time 109
7.7 Experimental Results 110
7.8 Conclusions 116

Summary of Notation **117**

Bibliography **119**

CHAPTER

1

Introduction

The title of this thesis, "Data Structures for Efficient String Algorithms," already suggests that this work is situated between two main areas of computer science: *data structures* and *string algorithmics*, the latter being meant to include both *string matching* and *string mining*. However, although almost all of our results on data structures do have applications in string matching, these two terms should not exclusively be seen in conjunction, as more than half of the present text is devoted to a problem whose applicability goes far beyond the area of stringology.

The main problem tackled in the first part of this thesis (chapters 3 and 4), which also gives the link between the different chapters, is so basic that it can easily be formulated already at this point: preprocess a one- or two-dimensional array of numbers such that queries asking for a minimum element between specified indices can be answered efficiently. Queries of this form are commonly called *Range Minimum Queries*, and given the natural formulation of this problem, it is not surprising that there is a wide range of algorithmic problems which have solutions with such queries at their heart.

Nevertheless, although there are applications of range minimum queries from many different fields, it must be said that all new results presented in the second part of this thesis (chapters 5–7) *do* come from the field of string matching and string mining. As range minimum queries or variants thereof form an essential part in most of our new methods, this only confirms the fact that they play an important role in state-of-the-art string algorithms.

This thesis follows a recent trend in data structures (and especially in text indexing), which is the research on *succinct* data structures. Although there is no clear mathematical definition of this term, we follow the common practice to call a data structure *succinct* if its space consumption in *bits* is linear in

the input size n, instead of the usual $O(n)$ *words* occupied by its non-succinct counterpart (which amounts to $O(n \log n)$ bits). The work on succinct data structures was initiated by Jacobson's succinct representation of unlabeled trees and graphs (1989), and since then a flood of results has appeared, including succinct encodings for ordered sets (Pagh, 2001), permutations (Munro et al., 2003), functions (Munro and Rao, 2004), labeled trees (Ferragina et al., 2005), text indexes (Ferragina and Manzini, 2005), range sums (Poon and Yiu, 2005), and strings (Sadakane and Grossi, 2006). This list is far from being complete, but already the examples given here and their recency of publication show that succinct data structures are a very active field of research.

A primary focus of this thesis lies on the *direct* construction of data structures. The term *direct* is borrowed from the realm of suffix arrays and usually reflects the fact that an array-based data structure can be computed without the help of dynamic data structures such as dynamically changing graphs or trees. This is an important concern, both in theory and practice. On the practical side, the peak space consumption at construction time is usually the bottleneck for the applicability of a method. But dynamic data structures tend to have a larger overhead than arrays in the space they consume; e.g., storing a sorted set of n numbers in an array just needs n words, whereas storing the same information in a dynamic search tree needs at least twice as much space, the n additional words accounting for the pointer information. It is exactly this line of reasoning that drove several groups of computer scientists in 2003 to devise *direct* linear-time algorithms for constructing suffix arrays, although it is easy to read off the suffix array from the suffix *tree*, whose linear-time construction had been shown by Weiner 30 years before.

From a theoretical point of view, direct algorithms are especially important in the case of succinct data structures: the use of dynamic data structures at construction time would often result in a peak space consumption of $O(n \log n)$ bits, whereas the final result occupies only $O(n)$ bits — an unnecessary waste of memory that one should certainly try to avoid!

1.1 Synopsis

We will now step through the main contributions of this thesis. This section is primarily intended for people who are familiar with the field of string matching and want to get an overview over the most important results.

Having defined in chapter 2 the necessary data structures and algorithmic techniques that are needed throughout this thesis, chapters 3 and 4 provide a deep investigation of the *Range Minimum Query* problem (RMQ-problem for short), which is to preprocess an array of n numbers (or, more generally, objects from a totally ordered set) such that subsequent queries asking for the position of a minimum element between two specified indices can be answered efficiently. Our main result is the following:

Theorem 3.11. *An array of n numbers can be preprocessed in $O(n)$ time to a data structure of size $2n+o(n)$ bits such that range minimum queries can be answered in $O(1)$ time. The space consumption at construction time is $O(\log^3 n)$ bits, apart from the $2n+o(n)$ bits occupied by the final data structure.*

Construction- and query-time are certainly optimal, and Thm. 3.12 shows that the space consumption of $2n+o(n)$ bits is also optimal under a reasonable model of computation (based on the comparison model). We will subsequently see that our new representation of RMQ-information leads to improvements in computing lowest common ancestors in trees (Thm. 3.13) and longest common extensions in strings (Thm. 3.14), two very important algorithmic problems.

For compressible input arrays we show that the RMQ-information can be compressed as well:

Theorem 3.15. *An array of n numbers from a set of size σ can be preprocessed in $O(n)$ time into a data structure of size $nH_k + O\left(\frac{n}{\log n}(k\log\sigma + \log^2\log n)\right)$ bits, simultaneously over all $k \in o(\log n)$, such that range minimum queries can be answered in $O(1)$ time. The space consumption at construction time is $2n+o(n)$ bits.*

Here, H_k denotes the k'th-order empirical entropy of the array, which basically measures the compressibility of the input ($0 \leq H_k \leq \log\sigma$ is "small" for compressible sequences). The interesting point to note on the space bound of Thm. 3.15 is that it matches the currently best known results for storing the input array itself in compressed form, while still being able to access any $O(\log n)$ contiguous bits in constant time under the RAM model.

Chapter 4 considers the two-dimensional generalization of the RMQ-problem, leading to

Theorem 4.2. *For any $k > 1$ which may be constant or not, an $(m \times n)$-matrix can be preprocessed in $O(nm(k+\log\log\ldots\log(mn)))$ time (there are $k+1$ log's) such that the position of a minimum value in an axis-parallel query-rectangle can be obtained in $O(1)$ time, using $O(kmn)$ words of additional space. This converges towards an algorithm with $O(mn\log^*(mn))$ preprocessing time* and *space and $O(1)$ query time.*

Observe that k appears in preprocessing time and space of Thm. 4.2, but *not* in the query time. This is the result of preprocessing the input matrix for k levels (using $O(mn)$ words each), while still answering all queries as if they appear on the first level. In the special case of $k=2$, this leads to an algorithm with $O(nm\log\log\log(mn))$ preprocessing time, $O(mn)$ space and $O(1)$ query time — note that time is already quasi-optimal.

Chapter 5 shows that our preprocessing scheme for RMQ also reduces the space consumption of the Enhanced Suffix Array (ESA). The ESA is a collection of arrays that provide the same functionality as the well-known suffix trees, but has a much better practical performance, both in primary and secondary memory. Among other results, we show

Theorem 5.5. *For a text T of length n over an alphabet Σ there is a data structure occupying $2n + o(n)$ bits that, together with the suffix- and LCP-array for T, allows the retrieval of all occ positions where pattern $P \in \Sigma^\star$ occurs in T in $O(|P| \cdot |\Sigma| + occ)$ time, for* any *alphabet size $|\Sigma|$. This data structure can be constructed in $O(n)$ time, and the additional space needed at construction time is $O(\log^3 n)$ bits.*

Note that in the (at least theoretically) highly important case where the alphabet size is constant this yields *optimal* string matching times, as $O(|P| + occ)$ is already the time to read the input pattern, plus the time to return all occurrences. However, if the alphabet size is not a constant, we also show the following improvement, based on a variant of our RMQ-precomputation:

Theorem 5.9. *For a text T of length n over an alphabet Σ there is a data structure with space-occupancy of $\approx 2.54311\, n + o(n)$ bits that, together with the suffix array and the LCP-array for T, allows the retrieval of all occ occurrences of a pattern $P \in \Sigma^\star$ in T in $O(|P| \log |\Sigma| + occ)$ time, for* any *alphabet size $|\Sigma|$. This data structure can be constructed in $O(n)$ time, and the additional space at construction time is $o(n)$ bits.*

Chapter 6 considers data mining problems over strings. In particular, we will show how to extract from a database of strings all patterns (i.e., substrings) whose frequency satisfies a given number of constraints, again using RMQs:

Theorem 6.2. *For a constant number of databases of strings of total length n, all strings that satisfy a frequency-based criterion (e.g., frequent or emerging substrings) can be calculated in $O(n + s)$ time, solely by using array-based data structures occupying $O(n)$ words of additional space (apart from the output), where s is the total size of the strings that satisfy the criterion.*

In a final chapter we consider the problem of indexing texts that are logically divided into *words*. We will show that if one is just interested in finding word-aligned occurrences (e.g., the search pattern "other" does not give a hit in "mother"), there is an analog to the suffix array which can be computed directly in optimal time and space:

Theorem 7.1. *Given a text T of length n consisting of k words (in the sense of natural or artificial languages) over an integer-alphabet, the* word suffix array *for T can be constructed directly in optimal $O(n)$ time, using only 3 integer arrays of length k apart from the text. The word suffix array occupies k (computer-)words of space.*

With the help of the word suffix array one gets the following matching times (in analogy to the full-text suffix array):

Theorem 7.6. *The number of word-aligned occurrences of a pattern P in a text T consisting of k words can be found in alphabet-independent $O(|P| \log k)$ time using the word suffix array, or in $O(|P| + \log k)$ time with the help of an additional array of size k. Text-independent string matching can be done in $O(|P| \log |\Sigma|)$ time, using another structure of size $O(k/\log k)$ words in addition to the two arrays from the $O(|P| + \log k)$ algorithm.*

1.2 Practicability of the Results

The previous section described the theoretical results of this thesis. Unfortunately, theoretical improvements do not always go hand in hand with practical improvements, but in our case they do: most of the results presented here improve previous approaches also in practice, sometimes even very drastically by several orders of magnitude. This will be confirmed through extensive tests of our methods on biological and other real-world or artificial data sets. Sect. 3.8 shows that the RMQ-preprocessing from Thm. 3.11 is very effective in practice, both in terms of time and space. The practical relevance of our string miner (Thm. 6.2) is given by the fact that it is the first method applicable to realistically-sized instances, as all previous approaches are either too slow, or consume too much memory to be useful in practice (see Sect. 6.4). Finally, the results on word suffix arrays (Theorems 7.1 and 7.6) are shown to be practically relevant as well in Sect. 7.7.

Two comments are in order at this place. The first is on the choice of methods that are used for practical comparisons. The author has decided to include only methods that have implementations which are released by their respective authors. This was ensured by either downloading the sources from their websites, or by contacting the authors directly. The reason for not coding "foreign" methods by oneself is simply to avoid the potential objection of having implemented the methods in a particularly "bad" way in order to advertise one's own methods. The only exception from this rule we made was for Alstrup et al.'s RMQ-method, as its description is already very low-level and can hence be translated one-to-one into source code, without leaving much room for bad choices of additional data structures and the like. The second comment is that the practical evaluation of some methods (in particular, Theorems 3.15, 5.5, and 5.9) has been given to students at Munich University as projects, and most of this work is still in progress.

1.3 Previous Publications

Parts of this thesis have already been presented at the 17th Annual Symposium on Combinatorial Pattern Matching in Barcelona, Spain (Fischer and Heun, 2006), the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases in Berlin, Germany (Fischer, Heun, and Kramer, 2006), the 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies in Hangzhou, China (Fischer and Heun,

2007), and the 18th Annual Symposium on Combinatorial Pattern Matching in London, Ontario (Ferragina and Fischer, 2007; Amir, Fischer, and Lewenstein, 2007).

1.4 A Note on Citation Policy

The author has decided to give priority to journal versions over conference contributions of cited results. There are several reasons for doing so; the main of these being the fact that journal versions usually have a higher degree of completeness in exposition. The disadvantage of this policy is that it seemingly leads to chronological oddities, for example, when a conference version from 2004 (which has not yet appeared in a journal) improves a result from 2006 (which is the journal version of a paper presented at a conference in, say, 1999). Having this in mind, the reader should not be led to any confusions.

Results by other authors that are presented in a theorem-like form are consistently labeled "Proposition," whether they are used as lemmata or theorems, without any valuation of their significance.

A final remark is that for many results that belong to the standard repertoire of a computer scientist we just cite text books instead of digging out their original source, or do not even give any reference at all (for *very* common concepts like big-Oh-notation, bucket sort, ...).

CHAPTER

2

Basic Concepts

The aim of this chapter is to define important notations and to present previous results that will be needed throughout this thesis. It can be safely skipped by readers who feel familiar in the field of combinatorial pattern matching. For convenience, there is also a summary of non-standard notation at the end of this thesis (page 117).

2.1 Arrays and Strings

As already seen in the introductory chapter, the basic objects we deal with are arrays and strings, so it is worthwhile to spend some lines that put them on a firm basis. An *array* $A[1, n]$ is a contiguous memory area such that each of its elements $A[1], A[2], \ldots, A[n]$ can be obtained in constant time, provided that the binary representations of the numbers in A fit into a constant number of computer words. It is a basic fact from information theory that if the numbers stored in the array are in the range $[0 : x-1] := \{0, 1, \ldots, x-2, x-1\}$, exactly $n\lceil \log x \rceil$ bits are needed to store A in the Kolmogorov sense. Because we will make heavy use of this fact in various chapters of this thesis, we introduce the notation $O(n \cdot \log x)$ for the number of bits needed to store such an array A. For example, a normal integer array of size n which takes values up to n uses $O(n \cdot \log n)$ bits of space. Here and in the rest of this thesis, "log" denotes the logarithm to base 2, whereas "ln" denotes the natural logarithm. $A[i, j]$ denotes A's sub-array ranging from i to j for $1 \leq i \leq j \leq n$.

Now a *string* is nothing else but an array of integers from a certain range. However, because it is almost always natural to distinguish between arrays of integers and strings of characters from an alphabet, we stick to the convention

to use a reserved notation for strings, as explained now. Let Σ be a totally ordered (usually finite) set, called the *alphabet*. The elements in Σ are called *characters*, and the order on the characters is usually denoted by "$<$." W.l.o.g. we can assume that $\Sigma = [1 : |\Sigma|]$, where $|\circ|$ denotes normal set size. A *string* S over Σ is then defined as a sequence of characters $a \in \Sigma$. $|S|$ denotes the *length* of S, i.e., the number of characters in S. The i'th character in S is denoted by S_i, starting at 1. For $1 \leq i \leq j \leq n$, we write $S_{i..j}$ to denote the substring of S ranging from position i to j. Σ^k is the set of length-k strings over Σ, $\Sigma^\star$ the set of all strings over Σ, and ϵ denotes the empty string (the string of length 0). For two strings $S, T \in \Sigma^\star$, ST denotes the *concatenation* of S and T.

We now extend the total order "$<$" on Σ to a *lexicographic order* on $\Sigma^\star$. Let $a, b \in \Sigma$ and $S, T \in \Sigma^\star$. Then $aS < bT$, in words "aS is lexicographic less than bT," if $a < b$, or if $a = b$ and $S < T$. The empty word ϵ is considered to be smaller than any non-empty string. We also use the notation $S \leq T$ to denote that either $S < T$ or $S = T$. For $S, T \in \Sigma^\star$ we write $S \trianglelefteq T$ if S is a non-empty substring of T, in symbols: $S = T_{i..j}$ for some $1 \leq i \leq j \leq |T|$. $S \sqsubseteq T$ denotes that S is a *prefix* of T, i.e., $S = T_{1..x}$ for some $1 \leq x \leq |T|$. $S \sqsubset T$ denotes a proper prefix. Finally, a *suffix* of a string S is any string of the form $S_{i..|S|}$ for some $1 \leq i \leq |S|$.

Example 2.1. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Then $\mathtt{a} < \mathtt{bc} \sqsubseteq \mathtt{bcba}$ and $\mathtt{b} \trianglelefteq \mathtt{aba}$.

2.2 Basic String Matching Problems

Let us now introduce some important classical problems on strings that are considered in this thesis.

Problem 2.1 (String Matching). For a given pattern P of length m and a (usually long) text T of length n, let $O_P \subseteq [1 : n]$ be the set of positions where P occurs in T: $i \in O_P$ iff $P \sqsubseteq T_{i..n}$. Then the tasks of string matching are (1) to answer whether or not O_P is empty *(decision query)*, (2) to return the size of O_P *(counting query)*, and (3) to enumerate the members of O_P in some order *(enumeration query)*.

Given the many contexts in which problems of this kind naturally arise, it is not surprising that there exists a wealth of literature devoted to these tasks. For a good overview of the subject, see Gusfield (1997, Part I). The best solutions are $O(n+m)$-algorithms for all of the tasks in Probl. 2.1, which is certainly optimal, as this is already the time to read the input. However, in many situation the text is static and known in advance, and there are many matching-queries to be answered on-line. In this case it makes sense to spend some time on building an *index* on T in order to answer future queries faster. This approach will be considered in the next sections.

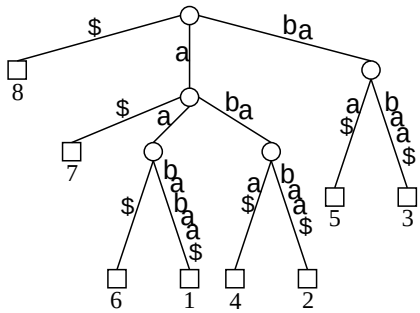


Figure 2.1: The suffix tree for $T =$ `aababaa`$.

2.3 Suffix Trees

Suffix trees are a well-known data structure for text indexing. Instead of defining them in a mathematical thorough way, we refer the reader to (Gusfield, 1997, Part II), and just describe them informally as follows: given a text $T = T_{1..n}$, a suffix tree for T is a rooted tree S whose edges are labeled with strings from $\Sigma^\star$, there is no node with out-degree 1, and no node has outgoing edges whose labels start with the same character (i.e., S is a compacted trie). The further condition is that each $P \trianglelefteq T$ can be "read" off the edges when walking down a path starting at the root of S. The name "suffix tree" is derived from the fact that if S is built for string $T' := T\$$, where \$ is a symbol not occurring elsewhere in T, then there is a one-to-one correspondence with between T's suffixes and the leaves in S (ignoring the single leaf representing "\$"). In this case we can also label the leaves with the position in T where the corresponding suffix starts. Fig. 2.1 shows an example for $T =$ `aababaa`$.

It is evident from the definition of suffix trees that decision-queries can be solved in $O(m)$ time by simply scanning the edges top-down and comparing their labels with subsequent letters from P, until P has either been completely matched, or the next symbol in P cannot be found. (Ignore for now the problem that at internal nodes some time is needed to find the correctly labeled outgoing edge.) Let v be the node where a successful matching procedure has brought us, or the first node below the point if we have end up on an edge. Then counting- and enumeration queries can be answered by a subsequent traversal of S_v, which denotes the subtree of S rooted at v. This takes $O(|O_P|)$ additional time, in the notation of Def. 2.1. This is optimal for enumeration queries, and a simple idea also yields optimal time bounds for counting queries: label the internal nodes with the size of their corresponding subtree; this lowers the time for counting queries to optimal $O(m)$ time.

There has been some confusion on the influence of the alphabet size, both on space usage and on preprocessing- and matching times. Let us first consider space occupation, which is directly connected to matching time. There are 3 basic implementation strategies for the outgoing edges of a node: (1) as arrays of size $|\Sigma|$, (2) as linked lists, usually sorted alphabetically, and (3) as balanced binary search trees. (1) has the advantage that matching time is truly

$O(m)$, without any influence of $|\Sigma|$, but the disadvantage is that space usage is $O(n|\Sigma|)$ words. For (2) and (3) it can be shown that the size of the suffix tree is $O(n)$ words, but matching times are $O(m|\Sigma|)$ and $O(m \log |\Sigma|)$, respectively. A good trade-off has been found by Munro et al. (2001), who enhance the $O(n)$-word representation of the suffix tree with an additional data structure of size $O(n \log |\Sigma|)$ *bits*, while supporting true $O(m)$ matching time.

For constant-sized alphabets it has been first shown by Weiner (1973) that suffix trees can be built it $O(n)$ time, and it took almost 30 years to achieve the same result for integer alphabets (Farach-Colton et al., 2000); i.e., an alphabet consisting of numbers from a discrete range of size $O(n)$.

2.4 Suffix- and LCP-Arrays

This section introduces two fundamental data structures that we will need at various places in this thesis.

The *suffix array* (Gonnet et al., 1992; Manber and Myers, 1993) for a given text T of length n is an array $\mathsf{SA}[1, n]$ of integers s.t. $T_{\mathsf{SA}[i]..n} < T_{\mathsf{SA}[i+1]..n}$ for all $1 \leq i < n$; i.e., SA describes the lexicographic order of T's suffixes by "enumerating" them from the lexicographically smallest to the largest. It follows from this definition that SA is a permutation of the numbers $[1 : n]$. For our example string $T = \texttt{aababaa\$}$, $\mathsf{SA} = [8, 7, 6, 1, 4, 2, 5, 3]$. Note that the suffix array is actually a "left-to-right" (i.e., alphabetical) enumeration of the leaves in the suffix tree for $T\$$. As SA stores n integers from the range $[1 : n]$, it takes n words (or $n \log n$ bits) to store SA in uncompressed form.

There are two basic search strategies for searching a pattern of length m with the help of a suffix array, both based on binary search (Manber and Myers, 1993). The first takes time $O(m \log n)$, and the second $O(m + \log n)$ with the help of another array occupying n words of space. In practice, these algorithms have proved to be competitive with the search in suffix trees (Manber and Myers, 1993). Sometimes, we will also need the *inverse suffix array* SA^{-1}, defined as $\mathsf{SA}[\mathsf{SA}^{-1}[i]] = i$ for all $1 \leq i \leq n$. In words, $\mathsf{SA}^{-1}[i]$ tells us where the i'th longest suffix can be found in SA.

As observed above, the suffix array can certainly be obtained from the suffix tree during a depth-first traversal of the latter. But there is also a wealth of literature on the *direct* construction of suffix arrays, starting with Manber and Myers's original $O(n \log n)$ method (1993) and culminating in three $O(n)$ algorithms needing just $O(n \log n)$ bits of additional working space (Kim et al., 2005; Ko and Aluru, 2005; Kärkkäinen et al., 2006), all of these working also for integer alphabets. There are also linear-time algorithms needing only $o(n \log n)$ bits of working space in addition to the final suffix array: for constant-sized alphabets, the algorithm due to Hon et al. (2003) uses $O(n)$ bits of space, and for general alphabets Na (2005) gave a linear-time method with a slightly higher space consumption. However, it has been noted by several teams of authors (e.g., Antonitio et al., 2004; Puglisi et al., 2005) that asymptotically

slower algorithms perform better in practice. For constant-sized alphabets, the methods due to Manzini and Ferragina (2004), Schürmann and Stoye (2007), and most recently also Maniscalco and Puglisi (2007) have been shown to be very efficient in practice. The first of these has the further advantage that it uses only $\epsilon < 1$ bytes of additional space per input character, which is, of course, close to optimal. Here, ϵ is a tunable parameter that determines the speed of the algorithm and can be made arbitrarily small. For integer alphabets, the $O(n \log n)$-method due to Larsson and Sadakane (1999) has a very good practical performance. Suffix sorting is still a popular area of research, as the appearance of two recent articles suggests (Franceschini and Muthukrishnan, 2007; Nong and Zhang, 2007). We refer the reader to Puglisi et al. (2007) who give a good overview of the advances made until 2005.

In the same way as suffix arrays store the leaves of the corresponding suffix tree S, the *LCP-array* captures information on the heights of the internal nodes in S as follows. $\mathsf{LCP}[1, n]$ is defined such that $\mathsf{LCP}[i]$ holds the length of the *longest common prefix* of the lexicographically $(i-1)$'st and i'th smallest suffixes. In symbols, $\mathsf{LCP}[i] = \max\{k : T_{\mathsf{SA}[i-1]..\mathsf{SA}[i-1]+k-1} = T_{\mathsf{SA}[i]..\mathsf{SA}[i]+k-1}\}$ for all $1 < i \leq n$, and $\mathsf{LCP}[1] = 0$.[1] For $T = \texttt{aababaa\$}$, $\mathsf{LCP} = [0, 0, 1, 2, 1, 3, 0, 2]$. Kasai et al. (2001) gave an algorithm to compute LCP in $O(n)$ time, and Manzini (2004) adapted this algorithm to work in-place, i.e., by using only $4n$ bytes on 32-bit computers.[2] It can be argued that most of the LCP-values are small compared to the size of the text and can thus be stored using less than $4n$ bytes, but we do not detail this approach here, as there is an even more space-efficient representation of LCP to be presented in Sect. 2.7.

2.5 Text Compressibility and Empirical Entropy

Consider a text $T_{1..n}$ with symbols from an alphabet Σ. In the Kolmogorov sense it takes $O(n \log |\Sigma|)$ bits to store T. However, it is obvious that texts exhibiting some kind of regularity can be stored in less space. Colloquially, one speaks of a good *compressibility* of T to reflect this fact. The *empirical entropy* puts this on a firm mathematical ground (Manzini, 2001):

Definition 2.1. *Let $T_{1..n}$ be a text with symbols from Σ. Then the* zeroth order empirical entropy *of T is defined as*

$$H_0 := H_0(T) := -\sum_{a \in \Sigma} \frac{n_a}{n} \log \frac{n_a}{n} \ ,$$

where n_a is the number times a occurs in T. $0 \log 0$ is defined as 0.

It is well known that $H_0 n$ is a lower bound on the size of the output of any compressor that assigns a fixed codeword to each symbol in T (Witten et al.,

[1] We will sometimes find it more convenient to define $\mathsf{LCP}[1] = -1$, but this should not lead to any confusion.

[2] Mäkinen (2003, Fig. 3) gives another algorithm to compute LCP almost in-place.

1999). If one takes into account a length-k context which precedes the symbol to be encoded next, the above definition generalizes to

Definition 2.2. *Let $T_{1..n}$ be a text with symbols from Σ. Then the k'th order empirical entropy of T is defined as*

$$H_k := H_k(T) := \frac{1}{n} \sum_{w \in \Sigma^k} |w_T| H_0(w_T) \ ,$$

where w_T denotes the string of symbols following the occurrences of w in T, reading T from left to right.

Thus, $H_k n$ is a lower bound for compressors who assign fixed code-words to symbols based on the preceding length-k context. We have the "hierarchy" $H_k \leq H_{k-1} \leq \cdots \leq H_0 \leq \log |\Sigma|$.

2.6 Compact Data Structures for Rank and Select

For many techniques referenced in this thesis it is important to know that there exist compact data structures for a bit-vector B of length n supporting the following operations for a fixed pattern $p \in \{0,1\}^\star$ in constant time:

- $\text{rank}_p(B, i)$: return the number of occurrences of pattern p in $B[1, i]$.
- $\text{select}_p(B, i)$: return the position of the i'th occurrence of pattern p in B.

It is immediate from this definition that these operations are the inverse of each other: $\text{rank}_p(B, \text{select}_p(B, i)) = i = \text{select}_p(B, \text{rank}_p(B, i))$. The following theorem is a combination of the results from Jacobson (1989), Munro (1996), and Clark (1996):

Proposition 2.3. *For a bit-vector B of length n and a constant-sized bit-pattern p, there exists a data structure for $\text{rank}_p(B, i)$ using $O(n \log\log n / \log n) = o(n)$ bits of space, and a data structure for $\text{select}_p(B, i)$ using $O(n \log^3 \log n / \log n) = o(n)$ bits of space.* ■

(The space for storing B itself in uncompressed form is n bits, of course.) We do not give the details on how these results are achieved; for short, they are a clever combination of multi-level storage schemes (Munro, 1996) and the so-called Four-Russians-Trick (Arlazarov et al., 1970) for precomputing all answers to sufficiently small sub-problems. We refer the reader to Navarro and Mäkinen (2007, Sect. 6) for a good exposition of these techniques. We finally remark that there are implementations for rank and select that work well in practice, though not guaranteeing the $O(1)$ lookup-times (González et al., 2005).

2.7 Succinct Representations of Suffix- and LCP-Arrays

A major drawback of suffix trees and arrays as described so far is that, because they employ pointers or store numbers up to the computer's word size, their space consumption is $O(n \log n)$ bits. As already mentioned in the introduction, it has been noted that in many cases this is far from optimal, and a flood of algorithms under the term *succinct* (meaning that their space complexity is (at least close to) $O(n)$ *bits* instead of *words*) has been developed, starting with Jacobson's succinct representation of unlabeled trees (1989). There are also succinct representations of the suffix- and the LCP-array which we will briefly survey here.[3]

There are several variants for representing the suffix array with less space, offering different tradeoffs between space and time.[4] The historical first of these are due to Mäkinen (2003) and Grossi and Vitter (2005). Both exploit regularities in the suffix array, in a similar way as Compact Directed Acyclic Word Graphs (Blumer et al., 1987) can be seen as a way to reduce the size of the suffix tree by merging isomorphic subtrees. The first of these is called the *Compact Suffix Array* and uses $2nH_k \log n + O(n \log \log n)$ bits of space. The time to access $\mathsf{SA}[i]$ is $O(\log^2 n / \log^2 \log n)$, although other space-time tradeoffs are possible. The second, called the *Compressed Suffix Array*, uses either $(1 + \epsilon^{-1})n \log |\Sigma| + 2n + O(n/\log \log n)$ or $(1 + \frac{1}{2} \log \log_{|\Sigma|} n)n \log |\Sigma| + 5n + O(n/\log \log n)$ bits, and access to $\mathsf{SA}[i]$ costs $O(\log^{\epsilon}_{|\Sigma|} n)$ or $O(\log \log_{|\Sigma|} n)$ time, respectively, for any fixed value $0 < \epsilon \leq 1$. There are two notable evolutions of the Compressed Suffix Array, the first due Sadakane (2003) who shows that the space of the first variant can actually bounded by $\epsilon^{-1} H_0 n + 2n + o(n)$ bits, and the second due to Grossi et al. (2003), bounding the space further to $H_k n + O(n \log \log n / \log n)$ bits, where looking up a value in the suffix array takes now $O(\log^2 n / \log \log n)$ time. Another interesting compressed representation of the suffix array is the *Succinct Suffix Array* (Mäkinen and Navarro, 2005), using $n(H_0 + 1)(1 + o(1))$ bits of space, while giving access to $\mathsf{SA}[i]$ in $O(H_0)$ *average* time. Again, we refer the reader to Navarro and Mäkinen (2007) for an excellent overview of this field.

Let us now come to the description of the succinct representation of the LCP-array due to Sadakane (2007a). The key to his result is the fact that the LCP-values cannot decrease too much if listed in the order of the inverse suffix array, a fact first proved by Kasai et al. (2001, Thm. 1):

Proposition 2.4. *For all* $i > 1$, $\mathsf{LCP}[\mathsf{SA}^{-1}[i]] \geq \mathsf{LCP}[\mathsf{SA}^{-1}[i-1]] - 1$. ■

Because $\mathsf{LCP}[\mathsf{SA}^{-1}[i]] \leq n - i + 1$ (the LCP-value cannot be longer than the length of the suffix!), this implies that $\mathsf{LCP}[\mathsf{SA}^{-1}[1]] + 1, \mathsf{LCP}[\mathsf{SA}^{-1}[2]] + 2, \ldots, \mathsf{LCP}[\mathsf{SA}^{-1}[n]] + n$ is an increasing sequence of integers in the range $[1, n]$.

[3] For lower bounds on the compressibility of suffix arrays, we refer the reader to the article by Schürmann and Stoye (2005).

[4] This section only reviews the approaches that give direct access to $\mathsf{SA}[i]$, although other indexes actually yield better string matching times (e.g., Ferragina and Manzini, 2005).

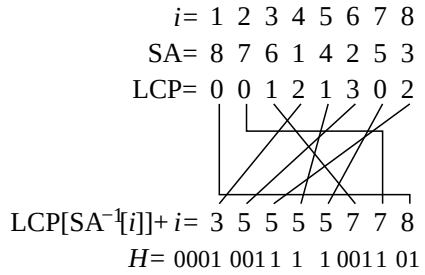


Figure 2.2: Illustration to the succinct representation of the LCP-array.

Now this list can be encoded *differentially*: for all $i = 1, 2, \ldots, n$, subsequently write the difference $\delta_i := \mathsf{LCP}[\mathsf{SA}^{-1}[i]] - \mathsf{LCP}[\mathsf{SA}^{-1}[i-1]] + 1$ of subsequent elements in unary code $u(\delta_i) := 0^{\delta_i}1$ into a bit-vector H, where we assume $\mathsf{LCP}[\mathsf{SA}^{-1}[-1]] = 0$. Here, 0^{δ_i} denotes the juxtaposition of δ_i zeros. Combining this with the fact that the LCP-values are all less than n, it is obvious that there are at most n zeros and exactly n ones in H. Further, if we prepare H for constant-time rank_0- and select_1-queries, we can retrieve $\mathsf{LCP}[i]$ as $\text{rank}_0(H, \text{select}_1(H, \mathsf{SA}[i])) - \mathsf{SA}[i]$. This is because the select_1-statement gives the position where the encoding for $\mathsf{LCP}[\mathsf{SA}[i]]$ ends in H, and the rank_0-statement counts the sum of the δ_j's for $1 \leq j \leq \mathsf{SA}[i]$. So subtracting the value $\mathsf{SA}[i]$, which has been "artificially" added to the LCP-array, yields the correct value. See Fig. 2.2 for an example.

This leads to

Proposition 2.5 (Succinct representation of LCP-arrays)**.** *The LCP-array for a text of length n can be stored in $2n + o(n)$ bits, while being able to access its elements in time proportional to the time needed to retrieve an element from the suffix array (i.e., in $O(1)$ time for uncompressed suffix arrays).* ■

We note that the succinct implementation can be computed in-place and in $O(n)$ time with an adaption of Kasai et al.'s algorithm (2001) for computing the LCP-array (see Hon and Sadakane, 2002).

2.8 Balanced Parentheses Representation of Trees

It is common folklore in computer science that the structure of a rooted tree T can be encoded by a sequence of *balanced parentheses* as follows: perform a depth-first traversal of T, writing an open parenthesis '(' if a node is visited for the first time, and a closing parenthesis ')' if it is visited for the last time (i.e., when all its subtrees have been traversed). The resulting sequence S is called *balanced parentheses sequence* or BPS for short. Because each node constitutes two bits in the BPS, the size of S is $2n$ bits for an n-node tree. As an example, the BPS of the suffix tree in Fig. 2.1 would be (()(()(()())(()())))(()())).

Munro and Raman (2001) show how most navigational operations in trees (such as finding the parent node, the i'th child, ...) can be simulated with the

BPS by preparing it for certain rank- and select-queries (see also Munro et al., 2001). For this thesis it is sufficient to know that a node in T can be uniquely identified by a pair of matching open and closing parentheses '(...)' in S. In fact, as the position of the closing parenthesis can be computed in $O(1)$ time from the position of the open parenthesis using a succinct dictionary of size $o(n)$ bits (Munro and Raman, 2001), we can also represent a node merely by the position of its corresponding '('.

We note that there is also an alternative succinct representation of trees called the *depth-first unary degree sequence* (Benoit et al., 2005). It also uses optimal $2n$ bits to represent a tree and offers a (partially complementary) set of navigational operations that it can simulate. The interested reader should consult a recent article (e.g., Jansson et al., 2007) for more details on this very active field of research.

CHAPTER

3

An Optimal Preprocessing Scheme for Range Minimum Queries

3.1 Chapter Introduction

This chapter deals with one of the most fundamental algorithmic problems in computer science: the task of preprocessing an array such that the position of a minimum value in a specified query interval can be obtained efficiently. Queries of this form are commonly called *Range Minimum Queries*, or just RMQ for short. Sect. 3.3 sketches a variety of problems that have RMQs at their heart, or are even equivalent to the RMQ-problem, such as the problem of finding lowest (or nearest) common ancestors of nodes in a tree. Sect. 3.4 then surveys the most important previous results on RMQ that we are aware of. We will see that the currently best solutions are either a *direct* algorithm due to Alstrup et al. (2002) using $O(n)$ words of space, or an *indirect* algorithm due to Sadakane (2007b) that uses $4n + o(n)$ bits of additional space for the final data structure, but $O(n)$ words at construction time. Recall from the introductory chapter that in analogy to direct construction of suffix arrays we call an algorithm "direct" if does not make use of dynamic data structures such as dynamic trees.

In Sect. 3.5 we give the first direct algorithm for RMQ that *never* uses more than $2n + o(n)$ bits, and further does not rely on other succinct data structures. In Sect. 3.6 we prove that our algorithm is asymptotically optimal, as there is a lower bound of $2n - o(n)$ bits under a reasonable model of computation. Sect. 3.7 sketches two important applications of our result, the problems of computing lowest common ancestors in trees and longest common extensions of suffixes. Sect. 3.8 shows that our new method and variants thereof are also of

practical value. Finally, we show in Sect. 3.9 that for compressible input arrays it is possible to reduce the space of our method further, namely to the same space that is needed by the currently best methods for storing the input array itself in compressed form, while still being able to access $O(\log n)$ contiguous bits in constant time.

3.2 Preliminary Definitions

The *Range Minimum Query* (RMQ)[1] problem is formally defined as follows: given an array $A[1, n]$ of elements from a totally ordered set (with order relation "$\leq$"), $\text{RMQ}_A(l, r)$ returns the index of a smallest element in $A[l, r]$, i.e., $\text{RMQ}_A(l, r) = \arg\min_{i \in [l:r]}\{A[i]\}$. (The subscript A will be omitted if the context is clear.) The most naive algorithm for this problem searches the array from l to r each time a query is presented, resulting in $O(n)$ query time. We consider the variant where the array is static and known in advance, and there are several queries to be answered on-line. In such cases it makes sense to preprocess A in order to answer future queries faster. Extending the notation from Bender et al. (2005), we say that an algorithm with preprocessing time $p(n)$ and query time $q(n)$ has *time*-complexity $\langle p(n), q(n)\rangle$. Thus, the naive method described above would be $\langle O(1), O(n)\rangle$, because it requires no preprocessing. The *space-consumption* is denoted as $[\![s(n), t(n)]\!]$, where $s(n)$ is the peak space consumption at construction time, and $t(n)$ is space consumption of the final data structure. In the rest of this chapter space is mostly analyzed in *bit*-complexity. Recall that for the sake of clarity we write $O(f(n) \cdot \log(g(n)))$ for the number of bits needed by a table of n positive integers, where $f(n)$ denotes the number of entries in the table, and $g(n)$ is their maximal size.

The following definition due to Vuillemin (1980) is central for the rest of this chapter (throughout this chapter, "binary" refers to trees with nodes having *at most* two children, and not *exactly* two).

Definition 3.1. *A* Cartesian Tree *of an array $A[l, r]$ is a binary tree $\mathcal{C}(A)$ whose root is a minimum element of A, labeled with the position i of this minimum. The left child of the root is the Cartesian Tree of $A[l, i-1]$ if $i > l$, otherwise it has no left child. The right child is defined similarly for $A[i+1, r]$.*

Note that $\mathcal{C}(A)$ is not necessarily unique if A contains equal elements. To overcome this problem, we impose a *strong* total order "$\prec$" on A by defining $A[i] \prec A[j]$ iff $A[i] < A[j]$, or $A[i] = A[j]$ and $i < j$. The effect of this definition is just to consider the "first" occurrence of equal elements in A as being the "smallest." Defining a Cartesian Tree over A using the $\prec$-order gives a *unique* tree $\mathcal{C}^{\text{can}}(A)$, which we call the *Canonical Cartesian Tree*. Note also that this order results in unique answers for the RMQ-problem, because the minimum is unique.

[1] Sometimes also called *Discrete Range Searching* (DRS) (Alstrup et al., 2002) or, depending on the context, Range Maximum Queries.

Gabow et al. (1984) give an algorithm for constructing $\mathcal{C}^{\text{can}}(A)$ which is summarized as follows. Let $\mathcal{C}_i^{\text{can}}(A)$ be the Canonical Cartesian Tree for $A[1,i]$. Then $\mathcal{C}_{i+1}^{\text{can}}(A)$ is obtained by climbing up from the rightmost leaf of $\mathcal{C}_i^{\text{can}}(A)$ to the root, thereby finding the position where $A[i+1]$ belongs. To be precise, let $v_1, \ldots, v_k$ be the nodes on the rightmost path in $\mathcal{C}_i^{\text{can}}(A)$ with labels $l_1, \ldots, l_k$, respectively, where v_1 is the root and v_k is the rightmost leaf. Let m be defined such that $A[l_m] \leq A[i+1]$ and $A[l_{m'}] > A[i+1]$ for all $m < m' \leq k$. To build $\mathcal{C}_{i+1}^{\text{can}}(A)$, create a new node w with label $i+1$ which becomes the right child of v_m, and the subtree rooted at v_{m+1} becomes the left child of w. This process inserts each element to the rightmost path exactly once, and each comparison removes one element from the rightmost path, resulting in a total $O(n)$ construction time to build $\mathcal{C}^{\text{can}}(A)$.

3.3 Applications of RMQ

We briefly sketch the most important applications of RMQ. Unless stated differently, the query time is $O(1)$. The preprocessing time is always $O(n)$, where n is the input size. Other applications of RMQ not mentioned here include finding common intervals in two or more sequences (Schmidt and Stoye, 2004), finding denominators in flow graphs (Georgiadis and Tarjan, 2004), and match-chaining algorithms (Shibuya and Kurochkin, 2003).

3.3.1 Computing Lowest Common Ancestors in Trees

The problem of finding the *Lowest Common Ancestor* (LCA)[2] of a pair of nodes in a tree has attracted much attention in the past three decades, starting with Aho et al. (1976). It is formally defined as follows: given a rooted tree T with n nodes, and two vertices v and w, find the deepest node $\text{LCA}_T(v,w)$ which is an ancestor of both v and w. Being one of the most fundamental problems on trees one can think of, LCA is not only algorithmically beautiful, but also has numerous applications in many fields of computer science, most importantly in the area of string processing and computational biology, where LCA is often used in conjunction with suffix trees. There are several variants of the problem, the most prominent being the one where the tree is static and known in advance, and there are several queries to be answered on-line (see Harel and Tarjan, 1984, for an overview of other variants). As for RMQs, in this case it makes sense to spend some time on *preprocessing* the tree in order to answer future queries faster. In their seminal paper, Harel and Tarjan (1984) showed that an intrinsic preprocessing in time *linear* in the size of the tree is sufficient to answer LCA-queries in *constant* time. Their algorithm was later simplified by Schieber and Vishkin (1988), but remained rather complicated.

A major breakthrough in practicable constant-time LCA-computation was made by Berkman and Vishkin (1993), and again, in a simplified presentation,

[2]The term *Nearest Common Ancestor* (NCA) is also frequently used for LCA.

by Bender et al. (2005). The key idea for this algorithm is the connection between LCA-queries on a tree T and RMQs on an array H derived from the *Euler-Tour* of T, first discovered by Gabow et al. (1984): let r be the root of T with k children $v_1, \ldots, v_k$. Then the Euler-Tour[3] of T is recursively defined as the array $E(T) = [v]$ if $k = 0$, and $E(T) = [v] \circ E(T_{v_1}) \circ [v] \circ E(T_{v_2}) \circ [v] \ldots [v] \circ E(T_{v_k}) \circ [v]$ otherwise, where "$\circ$" denotes array concatenation. In other words, E is an array of size $2n-1$, obtained by writing down the label of (or a pointer to) a node *each time* it is visited during a depth-first traversal of T. An additional array $H[1, 2n-1]$ holds the depths in T of the corresponding nodes in E. Finally, an array R of length n stores for each node v in T the position of a representative in E, i.e., $E[R[v]] = v$. We then have the following

Proposition 3.2 (Computing LCA via RMQ)**.** *Let T be a tree, E its Euler-Tour, H the array of heights derived from the Euler-Tour and R the representative array as defined above. Then $\text{LCA}_T(v, w) = E[\text{RMQ}_H(R[v], R[w])]$ for arbitrary nodes v and w.*

Proof. The elements in E between $R[v]$ and $R[w]$ are exactly the nodes encountered between v to w during a depth-first traversal of T, so the range minimum query returns the position k in H of the deepest such nodes. As the LCA of v and w must be encountered between v and w during the depth-first traversal, $\text{LCA}(v, w)$ is given by $E[k]$. ■

Gabow et al. (1984) also showed that the RMQ-problem can in turn be reduced to the LCA-problem by the use of Cartesian Trees, thus making the two problems linearly equivalent. However, as they could not *directly* solve RMQ-instances (i.e., without transformation to an LCA-instance), they could not use RMQ to improve on constant-time LCA-computation. As already mentioned above, this step was made by Berkman and Vishkin (1993). They noted that the reduction from LCA to RMQ via the Euler-Tour is in fact a reduction to a *restricted* version of RMQ, where consecutive array elements differ by exactly 1. We denote this restricted version by $\pm$1RMQ. Berkman and Vishkin (1993) then presented a *direct* algorithm for this restricted version of RMQ, which can then used to answer LCA-queries with the transformation described above. We will review their algorithm for $\pm$1RMQ in Sect. 3.4.1.

The drawback of the LCA-computation as presented in this section is the input doubling when going from T to E and H. In Sect. 3.7.1 we show that for binary trees and trees where the number of internal nodes is relatively close to the number of leaves this input doubling is not necessary.

3.3.2 Computing Longest Common Extensions of Suffixes

The problem of computing *Longest Common Extensions* (LCE), sometimes also denoted as the problem of *Longest Common Prefixes* (LCP), is defined

[3]The name "Euler-Tour" is derived from the Euler-Tour-technique by Tarjan and Vishkin (1985), and is not to be confused with an Eulerian circuit.

for a static string T of size n: given two indices i and j, $\text{LCE}_T(i,j)$ returns the length of the longest common prefix of T's suffixes starting at position i and j; i.e., $\text{LCE}_T(i,j) = \max\{k : T_{i..i+k-1} = T_{j..j+k-1}\}$.[4] This problem has numerous applications in exact and approximate pattern matching, e.g., for exact tandem repeats (Main and Lorentz, 1984; Gusfield and Stoye, 2004), approximate tandem repeats (Landau et al., 2001), inexact pattern matching (Myers, 1986; Landau and Vishkin, 1986), and direct construction of suffix trees or suffix arrays for integer alphabets (Farach-Colton et al., 2000; Kim et al., 2005).

It is well-known (e.g., see Gusfield, 1997) that $\text{LCE}_T(i,j)$ is given by the string-height of node $\text{LCA}_S(v_i, v_j)$ in the suffix tree S for T, where v_i and v_j are the leaves corresponding to suffix i and j, respectively. As the LCA-queries are only posed on the leaves of the suffix tree, it seems obvious that preparing the whole suffix tree for constant-time LCA-retrieval is quite an overhead. Fortunately, there exists a nice space-saving solution based on suffix arrays: Because of the one-to-one correspondence between S's leaves and the suffix array SA for T, and also between the heights of S's internal nodes and the LCP-array LCP for T, it is easy to see that $\text{LCE}(i,j) = \mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i]+1, \mathsf{SA}^{-1}[j])]$ (recall the definitions of SA and LCP in Sect. 2.4).

Note that for the LCE-algorithm based on RMQs it is crucial to use a *direct* algorithm for RMQ, for otherwise one could use a suffix tree in the first place and prepare it for LCA. Our new direct preprocessing scheme for RMQ has thus the consequence that most LCE-based algorithms cited above can be implemented without trees. However, it must be said that it is not immediately obvious how to do without trees in the Gusfield and Stoye's algorithm (2004) for computing tandem repeats, because it uses the tree structure also for representing the tandem repeats.

3.3.3 (Re-)Construction of Suffix Links

If it is necessary to augment a suffix tree with suffix links (e.g., when the suffix tree has been constructed from the suffix- and LCP-array), LCA-queries can be used for this task (see, e.g., Sect. 5.7.2 in Aluru, 2006). As these are in fact LCE-queries, with the algorithm from the previous paragraph this is another application of RMQ. Similarly, RMQs can be used directly to yield a linear-time algorithm to incorporate suffix links into the *Enhanced Suffix Array* (see Abouelhoda et al., 2004, Sect. 7). As the RMQs are again executed on the LCP-table, this is actually another special case of calculating the longest common extension of suffixes.

[4] LCE is often defined for *two* strings T' and T'' s.th. i is an index in T' and j in T''. This can be transformed to our definition by setting $T = T'\#T''$, where $\#$ is a new symbol.

3.3.4 Document Retrieval Queries

The setting of document retrieval problems is as follows: For a static collection of n text documents, on-line queries like "return all d documents containing pattern P" are posed to the system. Muthukrishnan (2002) gave elegant algorithms that solve this and related tasks in optimal $O(|P| + d)$ time. The idea behind these algorithms is to "chain" suffixes from the same document and use RMQs to ensure that each document containing P is visited at most twice. Sadakane (2007b) and Välimäki and Mäkinen (2007) continued this line of research towards succinctness, again using RMQs.

3.3.5 Maximum-Sum Segment Queries

These queries are also known as *maximal scoring subsequence*-queries. Given a static array $A[1, n]$ of n real numbers, on-line queries of the form "return the sub-interval of $A[l, r]$ with the highest sum" are to be answered; i.e., $\text{MSSQ}(l, r)$ returns the index pair (x, y) such that $(x, y) = \arg\max_{l \le x \le y \le r} \sum_{i=x}^{y} A[i]$. This problem and extensions thereof have very elegant optimal solutions based on RMQs due to Chen and Chao (2004). The fundamental connection between RMQ and MSSQ can be seen as follows: compute an array $C[0, n]$ of prefix sums as $C[i] := \sum_{k=0}^{i} A[k]$ for $1 \le i \le n$ and $C[0] := 0$, and prepare C for range minimum and range maximum queries. Then if $C[x]$ is the minimum and $C[y]$ the maximum of all $C[i]$ in $C[l-1, r]$ and $x < y$, then $(x+1, y)$ is the maximum-sum segment in $A[l, r]$. The more complicated case where $x > y$ is also broken down to RMQs.

3.3.6 Lempel-Ziv-77 Data Compression

Given a string $T = T_{1..n}$, the *LZ77-decomposition* (Ziv and Lempel, 1977) of T is a factorization $T = w_1 w_2 \dots w_x$ such that for all $j \in [1 : x]$, w_j is either a single letter not occurring in $w_1 \dots w_{j-1}$, or the longest substring occurring at least twice in $w_1 \dots w_j$. This factorization can be encoded as a sequence of x integer tuples (p_j, l_j), where $w_j = T_{p_j}$ if $l_j = 0$ (the "new letter"-case), or $w_j = T_{p_j..p_j+l_j-1}$ otherwise. The LZ77-decomposition can be computed in $O(n)$ time with with the help of a suffix tree (Rodeh et al., 1981), or using T's suffix- and LCP-array (Abouelhoda et al., 2004). Very recently, Chen et al. (2007) have shown how to compute LZ77 from the suffix array SA alone, with the help of RMQ-information on SA. This works roughly as follows: suppose that T has already been parsed up to position $i-1$: $T_{1..i-1} = w_1 \dots w_{j-1}$. The next step is to identify the longest prefix w_j of $T_{i..n}$ that occurs at least twice in $w_1 \dots w_j$. To find this prefix, first determine the interval $[l, r]$ of T_i in SA; i.e., $T_{\mathsf{SA}[l]..n}, \dots, T_{\mathsf{SA}[r]..n}$ are exactly those suffixes of T that are prefixed by T_i. $[l, r]$ can be found with an arbitrary search-algorithm in suffix arrays. To determine whether T_i occurs previously in $w_1 \dots w_{j-1}$, check if $\text{RMQ}_{\mathsf{SA}}(l, r) < i$. If not, T_i is a new letter, so output the tuple $(i, 0)$. Otherwise, narrow the interval $[l, r]$ by finding all entries in SA that are prefixed by $T_{i..i+1}$ (again, with any

search algorithm in suffix arrays). Continue this process x times such that $[l, r]$ is the last interval with $m := \text{RMQ}_{\mathsf{SA}}(l, r) < i$; output the tuple $(\mathsf{SA}[m], x)$. Although the theoretical running time of this algorithm is $O(n \log n)$ if one uses the binary search algorithm in suffix arrays, the practical performance is reported to be much better (especially if there are relatively few factors), while being very space-conscious (Chen et al., 2007).

3.4 Previous Results on RMQs

This section describes previous approaches for preprocessing an array $A[1, n]$ such that it supports range minimum queries in $O(1)$ time. While some of these techniques will also be important for our new algorithm to be presented in Sect. 3.5, others are merely presented for completeness of exposition.

3.4.1 The Berkman-Vishkin Algorithm

This section describes the solution to the general RMQ-problem as a combination of the results obtained in Berkman and Vishkin (1993) and Gabow et al. (1984). We follow the presentation from Bender et al. (2005) who rediscovered and simplified the algorithm.

The basic building block is a method to answer RMQs in constant time using $O(n \log n)$ *words* of space, called the *sparse table algorithm*. This algorithm will be used to answer "long" RMQs, and is also a basic ingredient for all subsequent RMQ-algorithms.[5] The idea is to precompute all RMQs whose length is a power of two. For every[6] $1 \leq i \leq n$ and every $1 \leq j \leq \lfloor \log n \rfloor$ compute the position of the minimum in the sub-array $A[i, i + 2^j - 1]$ and store the result in $M[i][j]$. Table M occupies $O(n \log n)$ words and can be filled in optimal time by using the formula $M[i][j] = \arg\min_{k \in \{M[i][j-1], M[i+2^{j-1}][j-1]\}} \{A[k]\}$. To answer $\text{RMQ}(l, r)$, select two *overlapping* blocks whose length is a power of two that exactly cover the sub-array $A[l, r]$, and return the position where the overall minimum occurs. Precisely, let $h := \lfloor \log(r - l) \rfloor$. Then $\text{RMQ}(l, r) = \arg\min_{k \in \{M[i][h], M[j-2^h+1][h]\}} \{A[k]\}$.

We now come to the description of the complete RMQ-algorithm (from now on called Berkman-Vishkin algorithm), which starts with a reduction from the general RMQ-problem to ± 1RMQ. This is a special case of the RMQ-problem, where consecutive array elements differ by exactly 1. We say that such arrays exhibit the ± 1-property. The reduction starts by building the Canonical Cartesian Tree $\mathcal{C}^{\text{can}}(A)$ as shown in Sect. 3.2, because of the following property:

[5] Berkman and Vishkin (1993) use a slightly more complicated algorithm, which is, however, equivalent to the one presented here.

[6] For an array $X[1, n]$ the expression $X[i, j]$ should be understood as $X[i, \min\{j, n\}]$ throughout this thesis.

Proposition 3.3 (Computing RMQ via LCA)**.** *Let A be an array with elements from a totally ordered set and $\mathcal{C}^{\text{can}}(A)$ its Canonical Cartesian Tree. Then $\text{RMQ}_A(l,r)$ is given by the label of node $\text{LCA}_{\mathcal{C}^{\text{can}}(A)}(v,w)$, where v and w are the nodes corresponding to l and r, respectively.*

Proof. Immediate from the inductive definition of the Cartesian Tree. ■

Note that with Prop. 3.2, this actually implies that the RMQ-problem and the LCA-problem are *linearly equivalent*, in the sense that an instance of one problem can be transformed into an instance of the other in linear time. This means that, in principle, one could use an arbitrary LCA-algorithm on $\mathcal{C}^{\text{can}}(A)$ to answer RMQs on A. But there is an important fact in the reduction from Prop. 3.2 which we have not yet mentioned, and which will lead us to a much more practical algorithm for RMQ: the elements in the height array H exhibit the ± 1-property. Combining this fact with Prop. 3.2, we see that Prop. 3.3 actually says that $\text{RMQ}_A(l,r) = E[\pm 1\text{RMQ}_H(R[l],R[r])]$, with H, E and R as described in Sect. 3.3.1, for completeness repeated here: $E[1,2n-1]$ is the Euler-Tour and is obtained from a depth-first traversal of $\mathcal{C}^{\text{can}}(A)$, storing the label of a node in E each time it is visited (i.e., also between the children of a node). $H[1,2n-1]$ stores the heights of the respective nodes in E, and $R[1,n]$ is a *representative array* such that $R[i]$ is the position of the first occurrence of $A[i]$ in E. As the Cartesian Tree is not needed anymore once the arrays E, H and R are filled, it can then be deleted. Note in particular the doubling of the input when going from A to H; i.e., H has size $n' := 2n-1$.

To solve ± 1RMQ on H, partition H into blocks of size $\frac{\log n'}{2}$.[7] Define two arrays H' and B of size $\frac{2n'}{\log n'}$, where $H'[i]$ stores the minimum of the ith block in H, and $B[i]$ stores the position of this minimum in H.[8] Now H' is preprocessed using the sparse table algorithm, occupying $O(\frac{2n'}{\log n'}\log\frac{2n'}{\log n'}) = O(n)$ words of space. This preprocessing enables out-of-block queries (i.e., queries that span over several blocks) to be answered in $O(1)$. It remains to show how in-block-queries are handled. This is done with the so-called Four-Russians-Trick (Arlazarov et al., 1970) where one precomputes the answers to all possible queries when the number of possible instances is sufficiently small. The authors of Berkman and Vishkin (1993) noted that due to the ± 1-property there are only $O(\sqrt{n'})$ blocks to be precomputed: we can virtually subtract the initial value of a block from each element without changing the answers to the RMQs; this enables us to describe a block by a ± 1-vector of length $1/2\log n' - 1$, implying that there are only $2^{1/2\log n'-1} = O(\sqrt{n'})$ possible blocks. For each such block precompute all $\frac{1}{2}\frac{\log n'}{2}(\frac{\log n'}{2}+1)$ possible RMQs and store them in a table P of total size $O(\sqrt{n'}\log^2 n') = O(n)$ words. Because we will also need table P for succinct solutions to the RMQ-problem (e.g., in Sect. 3.4.3), we already note here that the space for P is actually $O(\sqrt{n'}\log^2 n' \cdot \log n') = o(n')$ bits. To index table P, precompute the *type* of each block and store it in array $T[1,\frac{2n'}{\log n'}]$. The block type is simply the binary number obtained by comparing

[7] For a simpler presentation we often omit floors and ceilings from now on.
[8] Actually, array H' is just conceptual, as $H'[i]$ is given by $H[B[i]]$.

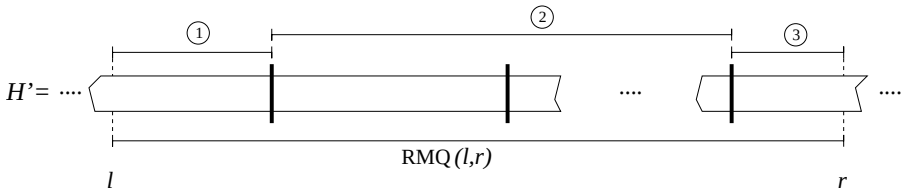


Figure 3.1: Decomposing $\text{RMQ}_{H'}(l, r)$ into three sub-queries. Vertical bars passing through H' indicate block-boundaries.

subsequent elements in the block, writing a 0 at position i if $H[i+1] = H[i]+1$ and 1 otherwise.

Now, to answer $\text{RMQ}(l, r)$, if l and r occur in different blocks, compute (1) the minimum from l to the end of l's block using arrays T and P, (2) the minimum of all blocks between l's and r's block using the precomputed queries on H' stored in table M, and (3) the minimum from the beginning of r's block to r, again using T and P. See also Fig. 3.1. Finally, return the position where the overall minimum occurs, possibly employing B. If l and r occur in the same block, just answer an in-block-query from l to r. In both cases, the time needed for answering the query is constant.

Because the algorithm needs several tables of size $O(n)$ storing numbers up to n, the space consumption of the algorithm is $[\![O(n \cdot \log n), O(n \cdot \log n)]\!]$. Time complexity is clearly $\langle O(n), O(1)\rangle$.

3.4.2 Alstrup et al.'s Idea for Handling In-Block-Queries

Alstrup et al. (2002) took a different approach to handle queries in a single block $B_j := A[(j-1)s+1, js]$ of size $s = O(\log n)$, for $j = 1, \ldots, n/s$. Instead of precomputing the answers to all possible queries and storing them in array $P[j]$ (thus using a total space of $O(s^2 \cdot \log s)$ bits for each block), they showed that $O(s \cdot s)$ bits are sufficient to allow a constant-time computation of all queries inside B_j.

The idea is to precompute for every position $i \in [1 : s]$ a bit-vector $V_j[i]$ of size s, where the kth bit of $V_j[i]$ is set to 1 iff $k < i$, $B_j[k] \leq B_j[i]$, but $B_j[k'] > B_j[k]$ for all $k < k' < i$. It is then easy to see that $\text{RMQ}_{B_j}(l, r)$ can be computed by clearing all bits with an index strictly smaller than l in $V_j[r]$ and returning the position of the least significant bit if the result is not 0; otherwise return r. Because the bit-vectors involved in these calculations are of size $s = O(\log n)$, these operations can be executed in $O(1)$ time under the RAM-model of computation.

Algorithm 3.1 computes the bit-vectors $V_j[1, s]$ for a block B_j in $O(s)$ time. The key insight is that for $i > 0$, $V_j[i]$ is obtained by setting the k'th bit in $V_j[k]$, where k is the largest index smaller than i with $B_j[k] \leq B_j[i]$ (line 7). A stack is used to allow an easy computation of this index k. After the while-loop in lines 3–4 the index k on top of the stack satisfies $B[k] \leq B[i]$, and because

Algorithm 3.1: Preprocessing for in-block-queries due to Alstrup et al.

Input: block B_j ranging from 1 to s
Output: array $V_j[1, s]$

```
1  let S be a stack, initially empty
2  for i ← 1, ..., s do
3      while notempty(S) ∧ B_j[i] < B_j[top(S)] do
4          pop(S)
5      endw
6      if empty(S) then  V_j[i] ← 0
7      else V_j[i] ← V_j[top(S)] OR (1 ≪ top(S))
8      push(S, i)
9  endfor
10 return V_j[1, s]
```

index i is pushed on the stack after the i'th iteration of the for-loop (line 8), k is the largest such index. Further, it is easy to verify that $V_j[1] = 0$ is also computed correctly.

One should note in particular that this preprocessing scheme also works for blocks that do not exhibit the ± 1-property and can thus be applied *directly* to the input array. This means that it is not necessary to go through the whole reduction chain via the Cartesian Tree in order to transform the original array into an array with the ± 1-property, as the Berkman-Vishkin algorithm does. However, because we have to precompute the bit-vectors for *all* $O(n/s)$ blocks, the total space needed for the precomputation of the in-block-queries is $O((n/s)s \cdot s) = O(n \cdot \log n)$ bits. Also, the sparse table algorithm from Sect. 3.4.1 is still necessary for the out-of-block queries, so the complete space consumption is $[\![O(n \cdot \log n), O(n \cdot \log n)]\!]$ bits.

3.4.3 Sadakane's Succinct RMQ-Algorithm

We now come to the description of a *succinct* data structure for RMQ due to Sadakane (2007b) which, like the algorithm from Sect. 3.4.1, also takes advantage of the interplay between LCA and RMQ, but with some subtle changes. The result is actually obtained in two steps. The first step (Sadakane, 2007b) is again a reduction from RMQs in an array A of size n to LCA-queries *on leaves* in an *extended* Cartesian Tree $\mathcal{C}^{\text{ext}}(A)$ with $n' := 2n$ nodes. The second step (Sadakane, 2007a) is a succinct data structure for constant-time LCA-retrieval in an arbitrary n'-node tree T using $o(n')$ extra bits — provided that T itself is already encoded as a balanced parentheses sequence (BPS) using $2n'$ bits. We already note here that due to the intermediate construction of the extended Cartesian Tree the peak space consumption of this algorithm is $O(n \log n)$, and as the BPS-encoding of a tree with $2n$ nodes takes $4n$ bits, the total space complexity is $[\![O(n \cdot \log n), 4n + o(n)]\!]$ bits. The details are as follows.

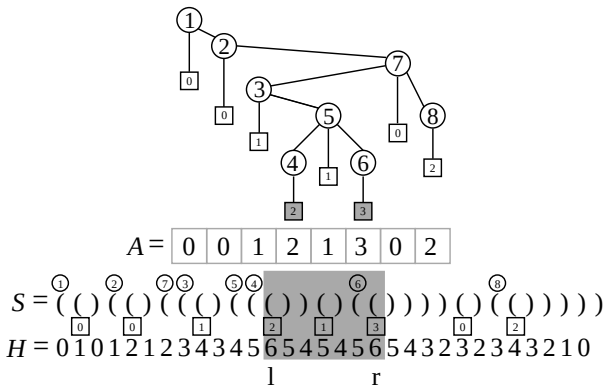


Figure 3.2: $\mathcal{C}^{\text{ext}}(A)$ for $A = [0, 0, 1, 2, 1, 3, 0, 2]$ (top) and its BPS S (bottom), where each '(' is labeled with its corresponding node. H contains the heights of the Euler Tour of $\mathcal{C}^{\text{ext}}(A)$. A query $\text{RMQ}_H(l, r)$ is shaded.

3.4.3.1 A Succinct Data Structure for LCA on BPS-encoded trees

Let us first describe Sadakane's algorithm (2007a) for the LCA-retrieval using $o(n')$ bits of additional space. Assume the tree T is encoded as a balanced parentheses sequence S as described in Sect. 2.8, and a node is identified with the position of its open parenthesis '('.

As in the case for the LCA-algorithm in Sect. 3.3.1, the overall aim is to be able to answer ± 1-RMQs on the height-array derived from the Euler-Tour of T (i.e., the array H in Sect. 3.3.1 and Sect. 3.4.1). Sadakane shows different ways how to compute the LCA of two nodes, somehow scattered over the articles (Sadakane, 2007a) and (Sadakane, 2007b); for our purposes, however, the following generalization of Prop. 3.2 suffices:

Proposition 3.4 (LCA-computation of leaves in BPS-encoded trees)**.** *For a tree T encoded by the balanced parentheses sequence S, let H be the height-array derived from a depth-first traversal of T. Then if $v \neq w$ are two leaves represented by the positions $l \leq r$ of their corresponding open parenthesis in S, $\pm 1\text{RMQ}_H(l, r)$ gives the position of the closing parenthesis of the first child-node of $\text{LCA}(v, w)$ that is visited between v and w in a depth-first traversal of T.*

Proof. From Prop. 3.2, $\text{RMQ}_H(l, r)$ yields the position p such that $H[p]$ equals the height of $u := \text{LCA}(v, w)$. During the depth-first traversal of T, node u must have been encountered when going "upwards" from a child c of u (note that $v \neq w$); thus in S one has written the closing parenthesis of c at position p. As RMQ always yields the leftmost minimum if this is not unique, c must be the first child of u. ■

As an example, look at the shaded query in Fig. 3.2. It asks for the LCA of the leaves labeled 2 and 3, respectively, and indeed, H attains the minimum in

the interval $[l, r]$ at the position of the closing parenthesis of the internal node labeled 4, the first child of the LCA of the two leaves.

Now here come the two crucial points which make linear bit-complexity possible: First, we have $H[i] = \text{rank}_((S, i-1) - \text{rank}_)(S, i-1)$, so array H need not be represented explicitly. This is simply because the number of '('s in S up to position $i-1$ tells us how often we have traversed an edge in T "downwards" when arriving at the node corresponding to position i in S, and likewise, the number of ')'s equals the number of edges that have been traversed "upwards;" thus, the difference gives the height of the node. And second, for indexing the table of the precomputed queries in blocks of size s (called "P" in Sect. 3.4.1), it is *not* necessary to store the types of the blocks! Recall that these block types were the sequence of $s-1$ bits obtained by writing a 0 into bit i if $H[i+1] = H[i]+1$ and a 1 if $H[i+1] = H[i]-1$; but this is exactly the part from the balanced parentheses sequence S corresponding to the block if one reads '(' as a 0 and ')' as a 1, and drops the first parenthesis. So S can be re-used for indexing table P of precomputed in-block-queries.

We are thus left with the task to show how ± 1-RMQs can be precomputed using only $o(n')$ extra space. Because table P occupies $o(n')$ bits of space, it only remains to show how the out-of-block queries can be stored succinctly. The idea is to introduce another preprocessing-layer with superblocks of size $s' = \log^3 n'$ and prepare both layers with the sparse-table algorithm from Sect. 3.4.1. We first wish to precompute the answers to all RMQs that span over at least one superblock. To do so, define a table $M'[1, n'/s'][0, \log(n'/s')]$. $M'[i][j]$ stores the position of the minimum in the sub-array $A[(i-1)s'+1, (i+2^j-1)s']$. As in Sect. 3.4.1, $M'[i][0]$ can be filled in a linear pass over the array, and for $j > 0$ we use a dynamic programming approach by setting $M'[i][j] = \arg\min_{k \in \{M'[i][j-1], M'[i+2^{j-1}][j-1]\}} \{A[k]\}$. In the same manner we precompute the answers to all RMQs that span over at least one block, but *not* over a superblock. These answers are stored in a table $M[1, n'/s][0, \log(s'/s)]$, where $M[i][j]$ stores the minimum of $A[(i-1)s+1, (i+2^j-1)s]$. Again, dynamic programming can be used to fill table M in optimal time.

Table M' has dimensions $n'/s' \times \log(n'/s') = n'/\log^3 n' \times \log(n'/\log^3 n')$ and stores values up to n'; the total number of bits needed for M' is therefore $n'/\log^3 n' \times \log(n'/\log^3 n') \cdot \log n' = o(n')$. Table M has dimensions $n'/s \times \log(s'/s) = 4n'/\log n' \times \log(\log^2 n')$ (remember that $s = \log n/2$, and the length of H is $2n'$). If we just store the offsets of the minima then the values do not become greater than s'; the total number of bits needed for M is therefore $O(n'/\log n' \times \log(\log^2 n') \cdot \log(\log^3 n')) = o(n')$.

3.4.3.2 A Different Reduction from RMQ to LCA

We now show how the general RMQ-problem can be reduced to an LCA-problem on leaves by incorporating the extended Cartesian Tree $\mathcal{C}^{\text{ext}}(A)$. This tree is obtained by taking the Canonical Cartesian Tree $\mathcal{C}^{\text{can}}(A)$ and adding a new leaf w as the *middle* child to each node v in $\mathcal{C}^{\text{can}}(A)$. Leaf w is labeled with

$A[i]$ if its parent node v is labeled with i. (Recall that the nodes in $\mathcal{C}^{\text{can}}(A)$ are labeled with the position of the minimum they represent.) $\mathcal{C}^{\text{ext}}(A)$ is then converted to a BPS S and can be deleted afterwards. Because $\mathcal{C}^{\text{ext}}(A)$ has $n' := 2n$ nodes (n internal nodes and n leaves), the size of S is $4n$ bits. See Fig. 3.2 for an example.

The reason for inserting the additional leaves in $\mathcal{C}^{\text{ext}}(A)$ is given by the following discussion: first observe that in a depth-first traversal of $\mathcal{C}^{\text{ext}}(A)$, the label of the ith visited leaf is $A[i]$; in other words, a depth-first traversal of $\mathcal{C}^{\text{ext}}(A)$ visits the leaves in the order of the original array A. Now Prop. 3.3 says that $\text{RMQ}_A(l, r)$ is given by the label of $\text{LCA}_{\mathcal{C}^{\text{ext}}(A)}(v, w)$, where v and w are the leaves in $\mathcal{C}^{\text{ext}}(A)$ corresponding to $A[l]$ and $A[r]$, respectively. But because leaves are represented by the sequence '()' in S, the index x in S of the open parenthesis of the leaf corresponding to $A[l]$ can be obtained by $x := \text{select}_{()}(S, l)$. Similarly, the index corresponding to $A[r]$ is $y := \text{select}_{()}(S, r)$. With Prop. 3.4, this means that $z := \pm 1\text{RMQ}_H(x, y)$ yields the position of the closing parenthesis corresponding to the first child c of $u := \text{LCA}_{\mathcal{C}^{\text{ext}}(A)}(v, w)$.

We are almost done. All that is left is to show how to obtain from z the index in A of $p := \text{RMQ}_A(l, r)$. The general idea is to first identify the position f in S of the open parenthesis of u's middle child, and then calculate this back to the final position p in A. Suppose first that v is a descendant of u's first child, which is c. In other words, z points to the position of the closing parenthesis of c. Then the next child must be the middle child represented by '()' in S, so we just need to check if $S[z + 1, z + 2] =$ '()' to see if we are in this case. If so, $f := z + 1$ gives the position of the open parenthesis of u's middle child. If, on the other hand, v is *not* a descendant of u's first child, c must already be u's middle child; in this case $f = z - 1$. Note that this can only happen if either v or w is already the middle child of u. Finally, because leaves are represented by '()' in S, the final index to be returned is obtained by $p = \text{rank}_{()}(S, f)$.

The advantage of this transformation is that because the parentheses sequence S captures both the height array and the block types, the original array A is not needed anymore once S has been created. Thus in applications where only the *position* of the minimum is of interest (and not the *value* itself) A can be deleted after preprocessing. However, the only such situation which we are aware of are document retrieval queries (Sect. 3.3.4).

3.5 An Improved Algorithm

We now come to the description of the main contribution of this chapter: a direct, practicable and optimal succinct representation of RMQ-information. One of the main building blocks of our solution is a new sequence for characterizing binary trees which we are going to describe first.

array A	$\mathcal{C}^{\text{can}}(A)$	path	$l_1 l_2 l_3$	number in enumeration
123			000	0
132			001	$C_{03} = 1$
231			002	$C_{03} + C_{02} = 2$
213			010	$C_{13} = 3$
321			011	$C_{13} + C_{02} = 4$

Table 3.1: Example-arrays of length 3, their Cartesian Trees, and their corresponding paths in the graph in Fig. 3.3. The last column shows how to calculate the index of $\mathcal{C}^{\text{can}}(A)$ in an enumeration of all Cartesian Trees.

3.5.1 A New Code for Binary Trees

This section introduces a new way to uniquely represent binary trees. Precisely, we will define a list (or sequence) of s numbers satisfying a certain "prefix property." This property will be very closely connected to Cartesian Trees and will thus be helpful for the improved RMQ-algorithm to be presented later. There are many other combinatorial objects which describe binary trees. E.g., the BPS of a tree from Sect. 2.8 is one such description. The ideas from this section can be seen as a new alternative for representing and enumerating binary trees. We refer the reader to Knuth (2006, Table 1 in Sect. 7.2.1.6) for other combinatorial objects describing binary trees. Throughout this section, the reader should peek to Tbl. 3.1, where most of the concepts are illustrated.

Let $l_1 l_2 \ldots l_s$ be a sequence of s natural numbers satisfying

$$\sum_{k=1}^{i} l_k < i \text{ for all } 1 \leq i \leq s \ . \tag{3.1}$$

For example, for $s = 3$, the five possible sequences are $a_1 = 000$, $a_2 = 001$, $a_3 = 002$, $a_4 = 010$, $a_5 = 011$. To derive all possible sequences for $s = 4$, one has to add one of the digits 0, 1, 2 and 3 to the right end of a_1, one of 0,1 and 2 to a_2 and a_4, and one of 0 or 1 to a_3 and a_5. This yields the 14 sequences

$$0000, 0001, 0002, 0003, 0010, 0011, 0012, 0020, 0021, 0100, 0101, 0102, 0110, 0111 \ .$$

How does such a sequence uniquely describe binary trees? First observe that each binary tree is a Canonical Cartesian Tree for some array A with elements from a totally ordered set of sufficient size; and by definition, each Canonical Cartesian Tree is also a binary tree. This implies that a binary tree T can, in principle, be represented by an array A: simply choose the numbers in A such that A's Canonical Cartesian Tree is equal to T.

The crucial fact to observe now is that the actual *numbers* in A do not affect the topology of the Cartesian Tree, as it is only determined by the *positions* of the minima. Recall the algorithm for constructing the Canonical Cartesian Tree in Sect. 3.2. In step i it traverses the rightmost path of $\mathcal{C}^{\text{can}}_{i-1}(A)$ from the rightmost leaf to the root and removes some elements from it. Now let l'_i be the number of nodes that are removed from the rightmost path when going from $\mathcal{C}^{\text{can}}_{i-1}(A)$ to $\mathcal{C}^{\text{can}}_{i}(A)$. Because one cannot remove more elements from the rightmost path than one has inserted before, and because each element is removed at most once, we have $\sum_{k=1}^{i} l'_k < i$ for all $1 \leq i \leq s$. Thus, the sequence $l'_1 \dots l'_s$ satisfying (3.1) completely describes the output of the algorithm for constructing the Canonical Cartesian Tree, and thus uniquely represents a binary tree with s nodes.

On the other hand, given a binary tree with s nodes, we can easily find a sequence of s numbers satisfying (3.1), by first constructing an array A whose Cartesian Tree equals the given tree, and then running the construction algorithm for $\mathcal{C}^{\text{can}}(A)$. In total, we have a bijective mapping from binary trees to sequences $l_1 l_2 \dots l_s$ satisfying (3.1).

Let $\mathcal{L}_s$ be the set of sequences $l_1 l_2 \dots l_s$ satisfying (3.1). Assume now we have defined a way to enumerate $\mathcal{L}_s$ in some order. Then for a given $l \in \mathcal{L}_s$ we might wish to compute the position (or index) of l in this enumeration. The most naive way to calculate the position of l in the enumeration would be to actually *construct* the Cartesian Tree related to l, and then use an inverse enumeration of binary trees (see Knuth, 2006) to compute its index. This, however, would counteract our aim to avoid dynamic data structures. We will show next how to compute this index directly. To prepare for an algorithm which solves this task, we first recall the definitions of some combinatorial numbers. It is well known that the number of binary trees with s nodes is the s'th *Catalan number* C_s, defined by $C_s := \frac{1}{s+1}\binom{2s}{s}$. By Stirling's approximation formula, we have

$$C_s = 4^s/(\sqrt{\pi} s^{3/2})\ (1 + O\left(s^{-1}\right))\ . \tag{3.2}$$

Closely related are the so-called *Ballot Numbers* C_{pq} (Knuth, 2006), defined by

$$C_{00} := 1, C_{pq} := C_{p(q-1)} + C_{(p-1)q},\ \text{if } 0 \leq p \leq q \neq 0\ , \tag{3.3}$$

and $C_{pq} := 0$ otherwise. It can be proved that a closed formula for C_{pq} is given by $\frac{q-p+1}{q+1}\binom{p+q}{p}$ (Knuth, 2006), which immediately implies that C_{ss} equals the s'th Catalan number C_s. If we look at the infinite directed graph shown in Fig. 3.3 then C_{pq} is clearly the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$, because of (3.3): if the current vertex is $\boxed{p\ q}$, one can either first go "up" and then take any of the $C_{p(q-1)}$ paths from $\boxed{p\ (q-1)}$ to $\boxed{0\ 0}$; or one first goes "left" to $\boxed{(p-1)\ q}$ and afterwards takes any of the $C_{(p-1)q}$ paths to $\boxed{0\ 0}$. The first few Ballot Numbers, laid out such that they correspond to the nodes in Fig.

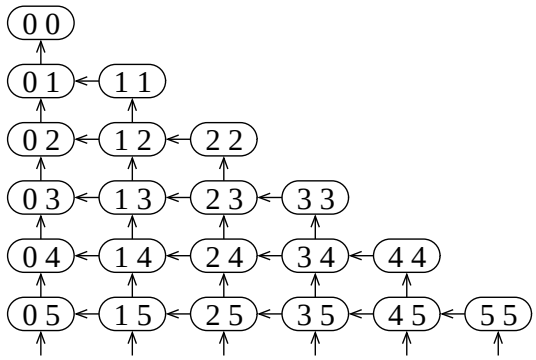


Figure 3.3: The infinite graph arising from the definition of the Ballot Numbers. Its vertices are $\boxed{p\ q}$ for all $0 \leq p \leq q$. There is an edge from $\boxed{p\ q}$ to $\boxed{(p-1)\ q}$ if $p > 0$ and to $\boxed{p\ (q-1)}$ if $q > p$.

3.3, are

$$\begin{array}{llllll} 1 & & & & & \\ 1 & 1 & & & & \\ 1 & 2 & 2 & & & \\ 1 & 3 & 5 & 5 & & \\ 1 & 4 & 9 & 14 & 14 & \\ 1 & 5 & 14 & 28 & 42 & 42\ . \end{array} \tag{3.4}$$

Now note that the sequence $l_1 \ldots l_s$ corresponds to a path from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in Fig. 3.3 (and vice versa). This is because the graph is constructed in a way such that one cannot move more cells upwards than one has already gone to the left if one starts at $\boxed{s\ s}$. So the path through the graph corresponding to $l_1 \ldots l_s$ is obtained as follows: in step i, go l_i steps upwards and one step to the left, and after step s go upwards until reaching $\boxed{0\ 0}$. Because there are C_s such paths, the task of computing the index of $l_1 \ldots l_s$ has thus become the task of finding a bijection from $\mathcal{L}_s$ to $\{0, 1, \ldots, C_s - 1\}$.

We now claim that the desired bijection is given by the function

$$f : \mathcal{L}_s \to \{0, 1, \ldots, C_s - 1\} : l_1 l_2 \ldots l_s \mapsto \sum_{i=1}^{s} \sum_{0 \leq j < l_i} C_{(s-i)(s-j-\sum_{k<i} l_k)}\ . \tag{3.5}$$

This formula is actually not so hard to understand when viewed from an algorithmic standpoint. The important thing to note is that it simulates a walk from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in the graph in Fig. 3.3. In step i of the outer sum, the current position in the graph is $\boxed{(s-i+1)\ (s-q)}$, with $q = \sum_{k<i} l_k$ being the total number of upwards steps that have already been made *before* step i. Now recall that l_i corresponds to moving l_i steps upwards, and then one step to the left. So for each of the l_i upward steps, the inner sum increments the value of the function by the number of paths that have been "skipped" by going upwards.

This is exactly $C_{(s-i)(s-q-j)}$, the value of the cell to the left of the current one if j runs through the upward steps. The effect of this incrementation is that paths going from the current position to the left are assigned lower numbers than paths going upwards. This implies that the sequence $l = 00\ldots0$ will be assigned the number $f(l) = 0$, $l' = 011\ldots1$ will get $f(l') = C_s - 1$, and other sequences will receive numbers between 0 and $C_s - 1$. As an example, the index of $l = 0102$ is $f(l) = C_{35} + (C_{14} + C_{13})$, the summand outside the parenthesis coming from $l_2 = 1$, and the two summands inside coming from $l_4 = 2$. See also Tbl. 3.1.

Let us now prove that the function f defined by (3.5) is actually bijective. From the discussion above we already know how we can bijectively map the sequences in $\mathcal{L}_s$ to paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in the graph in Fig. 3.3. Calling the set of such paths $\mathcal{P}_s$, we thus have to show that f is a bijection from $\mathcal{P}_s$ to $\{0, \ldots, C_s - 1\}$, with the intended meaning that the paths in $\mathcal{P}_s$ should actually first be mapped bijectively to a sequence in $\mathcal{L}_s$.

We need the following identities on the Ballot Numbers:

$$C_{pq} = \sum_{p \le q' \le q} C_{(p-1)q'} \text{ for } 1 \le p \le q \tag{3.6}$$

$$C_{(p-1)p} = 1 + \sum_{0 \le i < p-1} C_{(p-i-2)(p-i)} \text{ for } p > 0 \tag{3.7}$$

Eq. (3.6) follows easily by "unfolding" the definition of the Ballot Numbers, but before proving it formally, let us first see how this formula can be interpreted in terms of *paths*. (3.6) actually says that the number of paths from $\boxed{p\ q}$ to $\boxed{0\ 0}$ can be obtained by summing over the number of paths to $\boxed{0\ 0}$ from $\boxed{(p-1)\ q}$, $\boxed{(p-1)\ (q-1)}$, $\ldots$, $\boxed{(p-1)\ p}$, as all paths starting at $\boxed{p\ q}$ can be expressed as the disjoint union over those paths. The formal proof of (3.6) is by induction on q: for $q = 1$, $C_{11} = C_{10} + C_{01} = 0 + 1 = 1$ by (3.3), and (3.6) gives $C_{11} = \sum_{1 \le q' \le 1} C_{0q'} = C_{01} = 1$. For the induction step, let the induction hypothesis be $C_{p(q-1)} = \sum_{p \le q' \le q-1} C_{(p-1)q'}$ for all $1 \le p \le q-1$. Then

$$C_{pq} \overset{(3.3)}{=} C_{p(q-1)} + C_{(p-1)q} \overset{(\mathrm{IH})}{=} \sum_{p \le q' \le q-1} C_{(p-1)q'} + C_{(p-1)q} = \sum_{p \le q' \le q} C_{(p-1)q'} \ .$$

Eq. (3.7) is only slightly more complicated and can be proved by induction on p: for $p = 1$, $C_{01} = C_{00} + C_{(-1)1} = 1 + 0$ by (3.3), and (3.7) yields $C_{01} = 1 + \sum_{0 \le i < 0} C_{(-i-2)(-i)} = 1$, as the sum is empty. For the induction step, let the

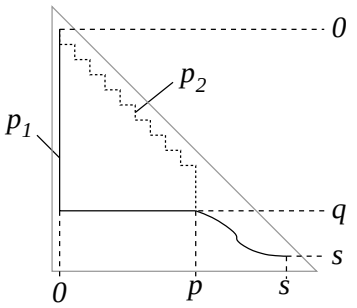


Figure 3.4: Smallest (p_1) and largest (p_2) paths (under f) among the paths that are equal up to $\boxed{p\ q}$.

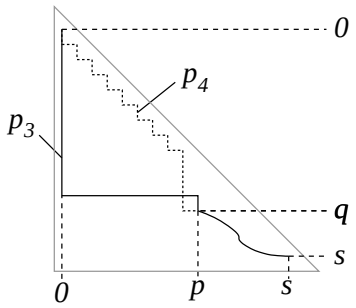


Figure 3.5: p_3 is the next-largest path (under f) after p_4 among those paths that are equal up to $\boxed{p\ q}$.

induction hypothesis be $C_{(p-2)(p-1)} = 1 + \sum_{0 \le i < p-2} C_{(p-1-i-2)(p-1-i)}$. Then

$$\begin{aligned}
C_{(p-1)p} &= C_{(p-1)(p-1)} + C_{(p-2)p} && \text{(by (3.3))}\\
&= C_{(p-1)(p-2)} + C_{(p-2)(p-1)} + C_{(p-2)p} && \text{(again by (3.3))}\\
&= C_{(p-2)(p-1)} + C_{(p-2)p} && \text{(because } C_{(p-1)(p-2)} = 0\text{)}\\
&= 1 + \sum_{0 \le i < p-2} C_{(p-1-i-2)(p-1-i)} + C_{(p-2)p} && \text{(induction hypothesis)}\\
&= 1 + \sum_{1 \le i < p-1} C_{(p-i-2)(p-i)} + C_{(p-2)p} && \text{(shifting indices)}\\
&= 1 + \sum_{0 \le i < p-1} C_{(p-i-2)(p-i)} - C_{(p-2)p} + C_{(p-2)p}\\
&= 1 + \sum_{0 \le i < p-1} C_{(p-i-2)(p-i)}\ .
\end{aligned}$$

We also need the following two lemmas for proving our claim.

Lemma 3.5. *For $0 \le p \le q \le s$ and an arbitrary (but fixed) path $\overline{pq}$ from $\boxed{s\ s}$ to $\boxed{p\ q}$, let $\mathcal{P}'_{\overline{pq}} \subseteq \mathcal{P}_s$ be the set of paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ that move along $\overline{pq}$ up to $\boxed{p\ q}$. Then f assigns the smallest number to the path $p_1 \in \mathcal{P}'_{\overline{pq}}$ that first goes horizontally from $\boxed{p\ q}$ to $\boxed{0\ q}$ and then vertically to $\boxed{0\ 0}$, and the largest value to the path $p_2 \in \mathcal{P}'_{\overline{pq}}$ that first goes vertically from $\boxed{p\ q}$ to $\boxed{p\ p}$, and then "crawls" along the main diagonal to $\boxed{0\ 0}$ (see also Fig. 3.4).*

Proof. The claim for p_1 is true because there are no more values added to sum when going only leftwards to the first column. The claim for p_2 follows from the left-to-right monotonicity of the Ballot Numbers, as seen in (3.4): $C_{ij} < C_{(i+1)j}$ for all $0 \le i+1 \le j-1$ (this follows directly from $C_{(i+1)j} = C_{ij} + C_{(i+1)(j-1)}$ and the fact that for $0 \le i+1 \le j-1$, $C_{(i+1)(j-1)} > 0$). So taking the rightmost

(i.e. highest) possible value from each row $q' \leq q$ must yield the highest sum (note that f can add at most one Ballot Number from each row $q' \leq q$). ■

Lemma 3.6. *Let p, q, and $\mathcal{P}'_{\overline{pq}}$ be as in Lemma 3.5. Let $p_3 \in \mathcal{P}'_{\overline{pq}}$ be the path that first moves one step upwards to $\boxed{p\ (q-1)}$, then horizontally until reaching $\boxed{0\ (q-1)}$, and then vertically to $\boxed{0\ 0}$. Let $p_4 \in \mathcal{P}'_{\overline{pq}}$ be the path that first moves one step leftwards to $\boxed{(p-1)\ q}$, then vertically until reaching $\boxed{(p-1)\ (p-1)}$, and then "crawls" along the main diagonal to $\boxed{0\ 0}$ (see also Fig. 3.5). Then $f(p_3) = f(p_4) + 1$.*

Proof. Let S be the sum of the Ballot Numbers that have already been added to the sum of both p_3 and p_4 when reaching $\boxed{p\ q}$. Then $f(p_3) = S + C_{(p-1)q}$, and

$$f(p_4) = S + \sum_{p \leq q' \leq q} C_{(p-2)q'} + \sum_{0 \leq i < p-2} C_{(p-i-3)(p-i-1)} \ ,$$

by simply summing over the Ballot Numbers that are added to S when making upwards moves.

Now

$$\begin{aligned}
f(p_3) &= S + C_{(p-1)q} \\
&= S + \sum_{p-1 \leq q' \leq q} C_{(p-2)q'} && \text{(by (3.6))} \\
&= S + \sum_{p \leq q' \leq q} C_{(p-2)q'} + C_{(p-2)(p-1)} \\
&= S + \sum_{p \leq q' \leq q} C_{(p-2)q'} + 1 + \sum_{0 \leq i < p-2} C_{(p-i-3)(p-i-1)} && \text{(by (3.7))} \\
&= f(p_4) + 1 \ .
\end{aligned}$$

■

This gives us all the tools for

Lemma 3.7. *Function f defined by (3.5) is a bijective mapping from $\mathcal{L}$ to $\{0, \ldots, C_s - 1\}$.*

Proof. Injectivity can be seen as follows: different paths p_3 and p_4 must have one point $\boxed{p\ q}$ where one path (w.l.o.g. p_3) continues with an upwards step, and the other (p_4) with a leftwards step. Combining Lemmas 3.5 and 3.6, f assigns a larger value to p_3 than to p_4, regardless of how these paths continue afterwards.

Surjectivity follows from the fact that the smallest path receives number 0, and the largest path (crawling along the main diagonal) receives number $\sum_{0 \leq i < s-1} C_{(s-i-2)(s-i)} = C_{(s-1)s} - 1$ (by (3.7)). But $C_{(s-1)s} = C_{ss} = C_s$,

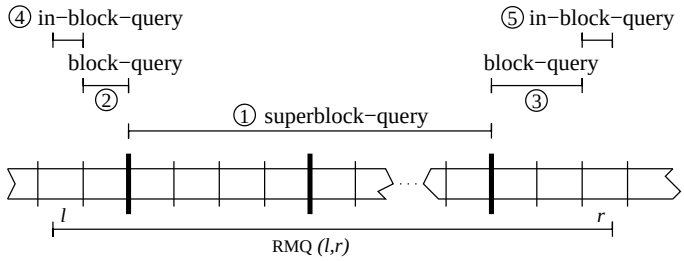


Figure 3.6: How a range-minimum query $\text{RMQ}(l, r)$ can be decomposed into at most five different sub-queries. Thick lines denote the boundaries between superblocks, thin lines denote the boundaries between blocks.

and as there are exactly C_s paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$, all being assigned different numbers, there must be a path $p \in \mathcal{P}$ such that $f(p) = x$ for every $x \in \{0, \ldots, C_s - 1\}$. ■

We summarize this section in the following

Lemma 3.8 (Enumeration of binary trees)**.** *For a binary tree T with s nodes, with* (3.5) *we can compute in $O(s)$ time the index of T in an enumeration of all binary trees with s nodes, with the help of a sequence of s numbers $l_1, \ldots, l_s$ defined by* (3.1)*.* ■

3.5.2 The Algorithm

This section describes our new algorithm for the RMQ-problem. The array A to be preprocessed is (conceptually) divided into superblocks $B'_1, \ldots, B'_{n/s'}$ of size $s' := \log^{2+\varepsilon} n$, where B'_i spans from $A[(i-1)s' + 1]$ to $A[is']$. Here, ε is an arbitrary constant greater than 0. Likewise, A is divided into (conceptual) blocks $B_1, \ldots, B_{n/s}$ of size $s := \log n/(2 + \delta)$. Again, $\delta > 0$ is a constant. For the sake of simplicity we assume that s' is a multiple of s. The general idea is that a query from l to r can be divided into at most five sub-queries (see also Fig. 3.6): one *superblock-query* that spans several superblocks, two *block-queries* that span the blocks to the left and right of the superblock-query, and two *in-block-queries* to the left and right of the block-queries. We will preprocess long queries by a two-level step as presented in Sect. 3.4.3.1, and short queries will be precomputed by a combination of the Four-Russians-Trick (Arlazarov et al., 1970) with the method from Sect. 3.4.2. From now on, we assume that the $\prec$-relation is used for answering RMQs, such that the answers become unique.

3.5.2.1 A Succinct Data Structure for Handling Long Queries

We first wish to precompute the answers to all RMQs that span over at least one superblock. In essence, we re-use Sadakane's idea for the succinct computation

of LCAs in BPS-encoded trees (see Sect. 3.4.3). This is actually a two-level storage scheme due to Munro (1996). Define a table $M'[1, n/s'][0, \log(n/s')]$. $M'[i][j]$ stores the position of the minimum in the sub-array $A[(i-1)s'+1, (i+2^j-1)s']$. As in Sect. 3.4.1, $M'[i][0]$ can be filled by a linear pass over the array, and for $j > 0$ we use a dynamic programming approach by setting $M'[i][j] = \arg\min_{k \in \{M'[i][j-1], M'[i+2^{j-1}][j-1]\}} \{A[k]\}$.

In the same manner we precompute the answers to all RMQs that span over at least one block, but *not* over a superblock. These answers are stored in a table $M[0, n/s-1][0, \log(s'/s)]$, where $M[i][j]$ stores the minimum of $A[(i-1)s+1, (i+2^j-1)s]$. Again, dynamic programming can be used to fill table M in optimal time.

3.5.2.2 A Succinct Data Structure for Handling Short Queries

We now show how to store all necessary information for answering in-block-queries using tables of size $2n + o(n)$ in total. The key to our solution is the following lemma, which has implicitly been used already in the Berkman-Vishkin algorithm. (Recall that the $\prec$-relation is used for answering RMQs, such that the answers become unique.)

Lemma 3.9 (Relating RMQs and Cartesian Trees)**.** *Let A and B be two arrays, both of size s. Then $\text{RMQ}_A(i,j) = \text{RMQ}_B(i,j)$ for all $1 \le i \le j \le s$ if and only if $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$.*

Proof. It is easy to see that $\text{RMQ}_A(i,j) = \text{RMQ}_B(i,j)$ for all $1 \le i \le j \le s$ if and only if the following three conditions are satisfied:

1. The minimum under "$\prec$" occurs at the same position m, i.e., $\arg\min A = \arg\min B = m$.
2. For all $1 \le i \le j < m$, $\text{RMQ}_{A[1,m-1]}(i,j) = \text{RMQ}_{B[1,m-1]}(i,j)$.
3. For all $m < i \le j \le s$, $\text{RMQ}_{A[m+1,s]}(i,j) = \text{RMQ}_{B[m+1,s]}(i,j)$.

Due to the definition of the Canonical Cartesian Tree, points (1)–(3) are true if and only if the root of $\mathcal{C}^{\text{can}}(A)$ equals the root of $\mathcal{C}^{\text{can}}(B)$, and $\mathcal{C}^{\text{can}}(A[1, m-1]) = \mathcal{C}^{\text{can}}(B[1, m-1])$, and $\mathcal{C}^{\text{can}}(A[m+1, s]) = \mathcal{C}^{\text{can}}(B[m+1, s])$. As this is the definition of Cartesian Trees, this is true iff $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$. ■

This means that table P does not have to store the in-block-queries for all n/s *occurring* blocks, but only for $C_s = 4^s/(\sqrt{\pi}s^{3/2})(1 + O(s^{-1}))$ *possible* blocks. The total space occupied by P will be analyzed in Sect. 3.5.2.5.

3.5.2.3 Computing the Block Types

In order to index table P, it remains to show how to compute the types of the n/s blocks B_j occurring in A in linear time; i.e., how to fill an array $T[1, n/s]$

Algorithm 3.2: An algorithm to compute the type of a block B_j

Input: a block B_j of size s
Output: the type of B_j, as defined by Eq. (3.8)

```
1  let R be an array of size s + 1   {R stores elements on the rightmost path}
2  R[1] ← −∞
3  q ← s, N ← 0
4  for i ← 1, ..., s do
5      while R[q + i − s] > B_j[i] do
6          N ← N + C_{(s−i)q}                    {add number of skipped paths}
7          q ← q − 1                             {remove node from rightmost path}
8      endw
9      R[q + i + 1 − s] ← B_j[i]                 {B_j[i] is new rightmost leaf}
10 endfor
11 return N
```

such that $T[j]$ gives the type of block B_j. Lemma 3.9 implies that there are only C_s different types of blocks, so we are looking for a surjection

$$t : \mathcal{A}_s \to \{0, \ldots, C_s - 1\}, \text{ and } t(B_i) = t(B_j) \text{ iff } \mathcal{C}^{\text{can}}(B_i) = \mathcal{C}^{\text{can}}(B_j) \ , \quad (3.8)$$

where $\mathcal{A}_s$ is the set of arrays of size s. The reason for requiring that B_i and B_j have the same Canonical Cartesian Tree is that in such a case both blocks share the same RMQs. Algorithm 3.2 shows how to compute the block type by making use of the ideas from Sect. 3.5.1.

Lemma 3.10 (Correctness of Type-Computation)**.** *Algorithm 3.2 correctly computes the type of a block B_j of size s in $O(s)$ time, i.e., it computes a function satisfying the conditions given in* (3.8)*.*

Proof. Intuitively, Alg. 3.2 simulates the algorithm for constructing $\mathcal{C}^{\text{can}}(B_j)$ given in Sect. 3.2 and "implements" a function defined by (3.5). First note that array $R[1, s+1]$ simulates the stack containing the labels of the nodes on the rightmost path of the partial Canonical Cartesian Tree $\mathcal{C}_i^{\text{can}}(B_j)$, with $q + i - s$ pointing to the top of the stack (i.e., the rightmost leaf), and $R[1]$ acting as a "stopper." Now let l_i be the number of times the while-loop (lines 5–8) is executed during the ith iteration of the outer for-loop. Note that l_i equals the number of elements that are removed from the rightmost path when going from $\mathcal{C}_{i-1}^{\text{can}}(B_j)$ to $\mathcal{C}_i^{\text{can}}(B_j)$. So the sequence $l_1 l_2 \ldots l_s$ satisfies (3.1), and therefore uniquely characterizes $\mathcal{C}^{\text{can}}(B_j)$. As the additions performed in line 6 of Alg. 3.2 are exactly those defined by (3.5), the value computed by the algorithm is the index of $\mathcal{C}^{\text{can}}(B_j)$ in an enumeration of all binary trees with s nodes. We conclude that Alg. 3.2 computes a function defined by Eq. (3.8). ■

3.5.2.4 Answering Queries

Having precomputed all the necessary information as above, it is easy to see that $\text{RMQ}(l, r)$ is one of the following five positions (precisely, the position where A attains the minimum value):

- If the interval $[l : r]$ spans the superblocks $B'_{l'}, \ldots, B'_{r'}$, compute the position of the minimum as $\arg\min_{k \in \{M'[l'][x], M'[r'-2^x+1][x]\}} \{A[k]\}$, where $x = \lfloor \log(r' - l') \rfloor$.

- If the interval $[l : r]$ spans the blocks $B_{l_1}, \ldots, B_{r_1}$ to the left of superblock $B'_{l'}$, compute the position of the minimum in this interval as $\arg\min_{k \in \{M[l_1][x], M[r_1-2^x+1][x]\}} \{A[k]\}$, where $x = \lfloor \log(r_1 - l_1) \rfloor$.

- Same as the previous point, but for the blocks $B_{l_2}, \ldots, B_{r_2}$ to the right of $B'_{r'}$.

- Compute the positions of the minimum in the blocks to the left of B_{l_1} and to the right of B_{r_2}.

3.5.2.5 Space Analysis

Table M' has dimensions $n/s' \times \log(n/s') = n/\log^{2+\varepsilon} n \times \log(n/\log^{2+\varepsilon} n)$ and stores values up to n; the total number of bits needed for M' is therefore

$$\begin{aligned} |M'| &= \frac{n}{\log^{2+\varepsilon} n} \log\left(\frac{n}{\log^{2+\varepsilon} n}\right) \cdot \log n \\ &= \frac{n}{\log^{1+\varepsilon} n}(\log n - \log\log^{2+\varepsilon} n) \\ &= \frac{n}{\log^{\varepsilon} n}(1 - o(1)) \\ &= o(n) \ . \end{aligned}$$

Table M has dimensions $n/s \times \log(s'/s) = (2+\delta)n/\log n \times \log((2+\delta)\log^{1+\varepsilon} n)$. If we just store the offsets of the minima then the values do not become greater than s'; the total number of bits needed for M is therefore

$$\begin{aligned} |M| &= O\left(\frac{n}{\log n} \log(\log^{1+\varepsilon} n) \cdot \log(\log^{2+\varepsilon} n)\right) \\ &= O\left(\frac{n \log^2 \log n}{\log n}\right) \\ &= o(n) \ . \end{aligned}$$

To store the type of each block, array T has length $n/s = (2+\delta)n/\log n$, and because of Lemma 3.9, the numbers do not get bigger than $O(4^s/s^{3/2})$. This

means that the number of bits to encode T is

$$\begin{aligned} |T| &= \frac{n}{s} \cdot \log\left(O\left(\frac{4^s}{s^{3/2}}\right)\right) \\ &= \frac{n}{s} \cdot (2s - O(\log s)) \\ &= 2n - O\left(\frac{n \log\log n}{\log n}\right) \\ &= 2n - o(n) \ . \end{aligned}$$

Finally, it is possible to store table P with $o(n)$ bits: Lemma 3.9 implies that P has only $O(\frac{4^s}{s^{3/2}})$ rows, one for each possible block-type. For each type we need to precompute $\text{RMQ}(i, j)$ for all $1 \leq i \leq j \leq s$, so the number of columns in P is $O(s^2)$. If we use the method described in Sect. 3.4.2 to represent the answers to all RMQs inside one block, this takes $O(s \cdot s)$ bits of space for each possible block.[9] The total space is thus

$$\begin{aligned} |P| &= O\left(\frac{4^s}{s^{3/2}} s \cdot s\right) \\ &= O\left(n^{2/(2+\delta)} \sqrt{\log n}\right) \\ &= O\left(n^{1-\frac{1}{2/\delta+1}} \sqrt{\log n}\right) \\ &= o(n/\log n) \end{aligned}$$

bits.

Because the space consumption of table M is asymptotically higher than the second order term that is subtracted from the space of table T, the total space needed is $2n + o(n)$ bits. And because the peak space consumption at construction time is the same as that of the final data structure (apart from the negligible $O(\log n \log\log n)$ bits needed for array R, and the $O(\log^3 n)$ bits for the Ballot-Numbers in Alg. 3.2), we can now state the main result of this section:

Theorem 3.11 (Succinct representation of RMQ-information)**.** *For an array with n elements from a totally ordered set, there exists an algorithm for the RMQ-problem with time complexity $\langle O(n), O(1)\rangle$ and bit-space complexity $[\![2n + o(n), 2n + o(n)]\!]$.* ■

We finally note that our algorithm is also easy to implement on PRAMs (or real-world shared-memory machines), where with n/t processors the preprocessing runs in time $\Theta(t)$ if $t = \Omega(\log n)$, which is work-optimal. This is simply because the minimum-operation is associative and can hence be parallelized after a $\Theta(t)$ sequential initialization (Schwartz, 1980).

[9] We remark that the usage of Alstrup et al.'s method for the precomputation of the in-block queries would not be necessary to achieve the $o(n)$ space bound, but it certainly saves some space.

3.6 A Lower Bound

It is interesting that the leading term ($2n$ bits) in Thm. 3.11 comes from table T, i.e., from remembering the type of all blocks occurring in A. One can wonder if this is really necessary. This section proves that, asymptotically, one cannot do better than this. But we first have to think about a reasonable model of computation. To see why this is necessary, take, for example, an array A whose entries are only 0's and 1's. Then very little space is needed to answer RMQs on A in $O(1)$ time, as the array itself can be used for indexing the table of precomputed in-block-queries. Thus, table T from the previous section is not needed, so the additional space is $o(n)$ bits.

For proving the lower bound on space we assume that the array is *only* used for minimum evaluations. To model this situation, in analogy to the cell-probe model by Yao (1981), we introduce the so-called *min-probe model*, where we only count evaluations of $\arg\min\{A[i], A[j]\}$, and all other computations and accesses to additional data structures (but *not* to the input array A) are free. This model is actually quite similar to the comparison model which is used, e.g., for showing the $\Omega(n \log n)$ lower bound for sorting n objects. However, in the comparison model the decisions of the algorithm are only based on previous comparisons, whereas in our case we have a "background" data structure which can be queried a lot of times. Therefore, the introduction of a new model is necessary.

Theorem 3.12 (Lower bound for RMQ)**.** *For an array A of size n one needs at least $2n - o(n)$ additional bits to answer $\text{RMQ}_A(l, r)$ in $O(1)$ for all $1 \le l \le r \le n$ in the min-probe model.*

Proof. Let $\mathcal{D}_A$ be the additional data structure for an arbitrary array A. To evaluate an arbitrary $\text{RMQ}_A(l, r)$ in $O(1)$, the algorithm can make k argmin-evaluations (constant k). The algorithm's decision on which $i \in [l : r]$ is returned as the minimum is based on the outcome of the $K := 2^k$ possible outcomes of these argmin-evaluations, and possibly on other computations in $\mathcal{D}_A$.

Now suppose there are less than $C_n/(K+1)$ different such $\mathcal{D}_A$'s. Then by the pigeonhole-principle there exists a set $\{A_0, \ldots, A_K\}$ of $K + 1$ arrays of length n with $\mathcal{D}_{A_i} = \mathcal{D}_{A_j}$ for all i, j, but with pairwise different Cartesian Trees. As the RMQ-algorithm cannot differentiate between those arrays based on their additional data structure (because they are the same), it can return different answers to RMQs for at most K of these arrays. So for at least two of them (say A_x and A_y) it gives the same answer to $\text{RMQ}(l, r)$ for all l, r. But A_x and A_y have different Cartesian Trees, so there must be at least one pair of indices l', r' for which $\text{RMQ}_{A_x}(l', r') \neq \text{RMQ}_{A_y}(l', r')$. Contradiction.

So there must be at least $C_n/(K+1)$ different choices for $\mathcal{D}_A$; thus the space needed to represent $\mathcal{D}_A$ is at least $\log(C_n/(K+1)) = \log(C_n) - \log(K+1) = 2n - o(n) - O(1)$ bits in the Kolmogorov-sense. ■

This theorem implies that the algorithm from Sect. 3.5 is asymptotically optimal, as the ratio $(2n + o(n))/(2n - o(n))$ converges to 1.

3.7 Applications of the New Algorithm

Theorem 3.11 naturally improves the space consumption of *all* algorithms based on RMQs. This section details the improvement on two of the most important applications of RMQ. Apart from yielding simpler and less space-consuming methods, we will see in Sect. 3.8 that one can also expect improvements in running times.

3.7.1 LCAs in Trees with Small Average Degree

Recall the LCA-problem from Sect. 3.3.1. There, we showed that an LCA-query on T basically corresponds to a ± 1RMQ on the heights of the nodes visited during an Euler-Tour in T. Because the size of an Euler-Tour is exactly $2n - 1$, this leads to an input doubling. We show in this section that using the algorithm presented in Sect. 3.5 overcomes this problem for binary trees and trees with small average degree. The basic idea is that because our RMQ-preprocessing can be applied directly to any array (and not only to arrays with the ± 1-property as the Berkman-Vishkin algorithm), we do not have to store the complete Euler-Tour of the tree.

Let T be a rooted tree with n nodes. Instead of storing the Euler-Tour of T, store an *inorder tree walk* (Cormen et al., 2001) of T which is defined as follows: Let r be the root of T which has k children $v_1, \ldots, v_k$. Then the inorder tree walk I of T is recursively defined as the array $I(T) = [v]$ if $k = 0$, and $I(T) = I(T_{v_1}) \circ [v] \circ I(T_{v_2}) \circ [v] \ldots [v] \circ I(T_{v_k})$ otherwise, where "$\circ$" denotes array concatenation. The size m of array I is clearly in the range $[n : 2n - 1]$, attaining the minimum n for binary trees. We further, store the heights of each node in $H[1, m]$, i.e., $H[i]$ is the height of node $I[i]$ in T. Finally, let $R[1, n]$ be the representative array of I, i.e., $R[v]$ stores the position of an arbitrary occurrence of node v in I (for convenience, the first occurrence). The only difference between the Euler-Tour and the inorder tree walk is that the latter only stores references to the nodes *between* its children, whereas the former also inserts references to a node before its first and after its last child. Because the proof of Prop. 3.2 does not make use of these first and last references to a node, we see that we also have the connection $\text{LCA}_T(v, w) = I[\text{RMQ}_H(R[v], R[w])]$. Thus, preparing H directly for RMQ with the algorithm from Sect. 3.5 is enough for constant time LCA-computation in T.

The first improvement of this algorithm comes from the use of the inorder tree walk instead of the Euler Tour. If T is binary, the extra space needed is $3n$ words for the arrays I, H and R. Compare this with the LCA-algorithm based on the Euler-Tour which uses $5n$ words for the arrays E, H and R. For trees other than binary, the space for these three arrays is never more than $5n$

words; the reduction, however, is only relevant if the number of internal nodes is relatively close to the number of leaves (i.e., trees with small average degree).

The second improvement comes from using the new RMQ-algorithm presented in Sect. 3.5 which occupies only $O(n/\log n)$ extra words (Thm. 3.11). The only other two RMQ-algorithms which could also be applied directly to the inorder tree walk are those of Alstrup et al. and Sadakane. However, the former would result in at least n extra words (for the precomputation of the in-block-queries), and although the latter also needs $O(n/\log n)$ extra words, it has a much higher space consumption at construction time.

Theorem 3.13 (Improved constant-time LCA-computation)**.** *For a tree T with s nodes having at most 1 child, and l nodes $v_1, \ldots, v_l$ of out-degree $d(v_i) > 1$, there is a preprocessing scheme using $3(s - l + \sum_{i \in [1:l]} d(v_i)) + O(n/\log n)$ words which can be computed in-place in $O(n)$ time such that subsequent LCA-queries can be answered in constant time.* ■

3.7.2 An Improved Algorithm for Longest Common Extensions

Recall the LCE-problem from Sect. 3.3.2. There, we saw that $\textsc{lce}(i, j)$ can be obtained by $\mathsf{LCP}[\textsc{rmq}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i] + 1, \mathsf{SA}^{-1}[j])]$. We also discussed that it is crucial to use an RMQ-algorithm which does not make use of any dynamic data structures such as trees — for otherwise one could use a suffix tree in the first place and prepare it for constant time LCA-retrieval. Thus, the Sadakane-RMQ from Sect. 3.4.3 is not suitable for this task, because it constructs an intermediate extended Cartesian Tree and therefore needs $O(n \log n)$ bits of intermediate dynamic space. So our RMQ-algorithm is the best choice for this problem, if one wants to be more space-economical than Alstrup et al.'s solution 2002, which is the only other RMQ-algorithm which can be directly applied to LCP.

Nevertheless, it must be said that there exists a different solution to the LCE-problem which does not need dynamic data structures (Sadakane, 2007a, Thm. 2). This algorithm, however, makes use of the balanced parentheses representation S of the suffix tree, and some heavy algorithmic machinery is needed to construct S directly without first constructing the actual suffix tree. We briefly mention the two ideas for constructing S directly which we are aware of: The first (Hon and Sadakane, 2002) builds on the algorithm for simulating bottom-up traversals of suffix trees by means of the LCP-array (Kasai et al., 2001, Sect. 5). The second (Välimäki et al., 2007, Sect. 5) is an extension of the suffix-insertion algorithm due to Farach-Colton et al. (2000, Sect. 3); see also Crochemore and Rytter (2002, Thm. 7.5). As mentioned before, both algorithms are non-trivial, and as the BPS S of the suffix tree has size $4n$ bits, the space consumption of the resulting data structure for LCE is also twice as high as that of our algorithm. In any case, we get the following result:[10]

[10] Note that the statement of Thm. 3.14 does not immediately solve the LCE-problem as defined in Sect. 3.3.2, as there we need to have access to the *inverse* suffix array. However,

Theorem 3.14 (Space-efficient computation of LCE)**.** *After a linear-time preprocessing of a text $T_{1..n}$ resulting in a data structure of size $|\mathsf{SA}| + 4n + o(n)$ bits, the length of the longest common extension of the suffixes $T_{\mathsf{SA}[i]..n}$ and $T_{\mathsf{SA}[j]..n}$ can be computed in $O(t_{\mathsf{SA}})$ time, for any two indices i and j. Here, $|\mathsf{SA}|$ denotes the space of the suffix array, and t_{SA} denotes the time to access an element from SA. For the various space-time-tradeoffs between $|\mathsf{SA}|$ and t_{SA}, see Sect. 2.7.*

Proof. Follows directly from the fact that our preprocessing scheme for RMQ can also be applied to the succinct representation of the LCP-array (as described in Sect. 2.7). ■

This improves Thm. 2 of Sadakane (2007a) who achieves $|\mathsf{SA}| + 6n + o(n)$ bits of space within the same time bounds.

3.8 Practical Considerations

We implemented the preprocessing scheme from Sect. 3.5 in C++ (available at `www.bio.ifi.lmu.de/~fischer`) in the following three variants (optimized for 32-bit computing):

1. A *non-succinct* version using $4n + O(\sqrt{n \log n})$ *words* of additional space. It is a straightforward combination of the Berkman-Vishkin algorithm (Sect. 3.4.1) with our new preprocessing scheme for short queries (Sect. 3.5.2.2 and 3.5.2.3). The block-size was set to $s = \log n/4$. In-block-queries were precomputed with Berkman and Vishkin's method (*not* with Alstrup et al.'s method). The advantage over the Berkman-Vishkin algorithm is that there is no need for an intermediate construction of the Canonical Cartesian Tree, thereby also avoiding the input doubling. Further, arrays E, H and R from Sect. 3.4.1 are not needed at all. This implementation corresponds to the paper presented at CPM 2006 (see Fischer and Heun, 2006). It was just included to show that our succinct version below is better in terms of preprocessing time *and* space.

2. A *succinct* version as described in Sect. 3.5.2. A good trade-off between time and space is to fix the block-size s to 2^3 (implying that table P stores single bytes using Alstrup et al.'s method), and the superblock-size s' to 2^8 (implying that table M also stores single bytes). Additionally, we introduced an "intermediate" block-division of size $s'' = 2^4$. Then these intermediate blocks consist of two blocks of size $s = s''/2$; their RMQs can thus be answered with two look-ups to table P. This means that we do not have to store any information at all for the intermediate block-division. The advantage of this intermediate layer is that it reduces

as most compressed representations of the suffix array give access to its inverse within the same time bounds, the theorem can usually be seen as a solution to the LCE-problem.

method	space usage (bytes)	theor. query time
Berkman-Vishkin (BV)	$\leq 35n$[11]	$O(1)$
Alstrup et al.	$\leq 7n$	$O(1)$
non-succinct	$\leq 14n$	$O(1)$
succinct	$\leq \frac{7}{8}n$	$O(1)$
engineered	$\leq \frac{1}{4}n$	$O(\log n)$
naive	0	$O(n)$

Table 3.2: Different preprocessing schemes for RMQ used in our tests.

the space of table M'. In total, our implementation uses $\leq \frac{7}{8}n$ bytes if $n \leq 2^{32}$.

3. An *engineered* version where the superblock-size was set to $s' = 2^8$, and the block-size to $s = 2^6$. These two layers were preprocessed with the method from Sect. 3.5.2.1. Queries lying inside the blocks of size s were answered naively, i.e., by a simple linear scan for the minimum. This implementation uses $\leq \frac{1}{4}n$ bytes if $n \leq 2^{32}$. The choices for s and s' were made such that this method has a query time that is comparable to the succinct version above, while consuming less space.

We compared these implementations to the following three preprocessing schemes:

1. The Berkman-Vishkin algorithm (BV) as described in Sect. 3.4.1. This algorithm transforms the input array to a ± 1RMQ-instance by first building the Canonical Cartesian Tree. The block size was set to $s = \log n/2$.

2. *Alstrup et al.*'s algorithm as described in Sect. 3.4.2. The preprocessing for long queries is similar to BV, but short queries are handled differently. The block size was set to $s = 2^5$, as then the bit-vectors fit into a single computer word and can hence be manipulated quickly.

3. The *naive* algorithm which scans the query interval linearly each time an RMQ is posed. This method was just included to see for which query lengths the preprocessing for the other methods pays off.

See Tbl. 3.2 for an overview of the implementations used. Unfortunately, we could not compare to Sadakane's solution (Sect. 3.4.3). The reason for this is that there is no publicly available reference implementation. As the performance of Sadakane's method is heavily influenced by the implementation of the rank- and select-structures, and choosing the wrong such implementation would make our comparison very vulnerable, we opted for not implementing his method on our own.

[11] The intermediate space for constructing the Cartesian Tree is *not* included in the $35n$ bytes.

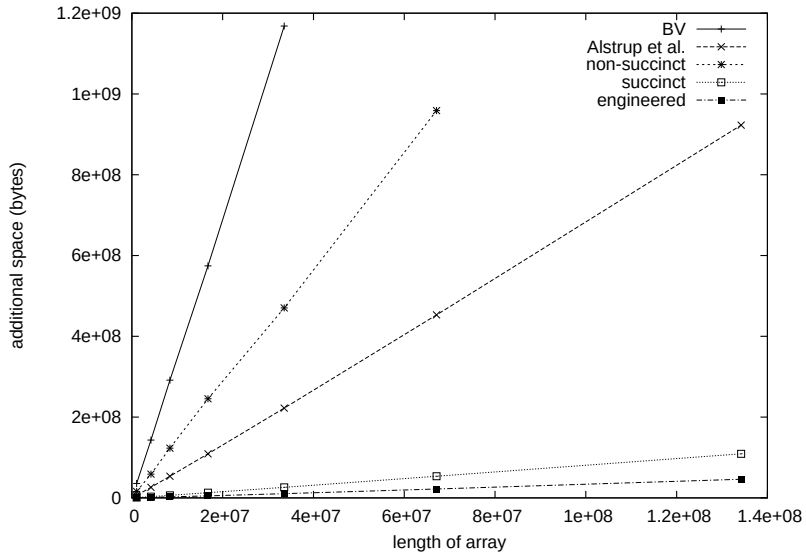


Figure 3.7: Final space consumption of different RMQ-schemes for varying array lengths. The space for constructing the Cartesian Tree (method BV) is not included in this graph.

All tests were performed on an Athlon XP 3300 with 2GB of RAM under Linux. All programs were compiled with `g++`, using the options "-O3 -fomit-frame-pointer -funroll-loops."

The implementations were tested on random input arrays of varying lengths. We emphasize the fact that both time and space of all implementations are largely independent of the input data, so there is no need to test the methods on different inputs. This fact was confirmed by conducting the same experiments on LCP-arrays derived from several test files from the the Pizza & Chili-site (Ferragina and Navarro, 2005), covering English texts and biological sequence data (DNA and proteins). But as the results for these files are similar to the ones presented below, we do not display them here.

We first evaluated the preprocessing time and space for varying array lengths from $n = 2^{20} \approx 10^6$ to $n = 2^{27} \approx 1.34 \times 10^8$. (So already the input array uses up to $4 \times 2^{27} = 512$MB of space.) First look at the space measurements in Fig. 3.7. This graph confirms the figures from Tbl. 3.2 and illustrates graphically the huge difference in space of the different methods. Note that for method BV the intermediate space for constructing the Cartesian Tree is not even included in the graph; doing so would certainly result in a more dramatic gap. This intermediate space is also the reason why we could test BV just on arrays up to length 2^{25}. Alstrup et al.'s method is the most space-conscious among the non-succinct schemes, but is beaten by orders of magnitude by our succinct implementation. As expected, the engineered version uses even less space than this.

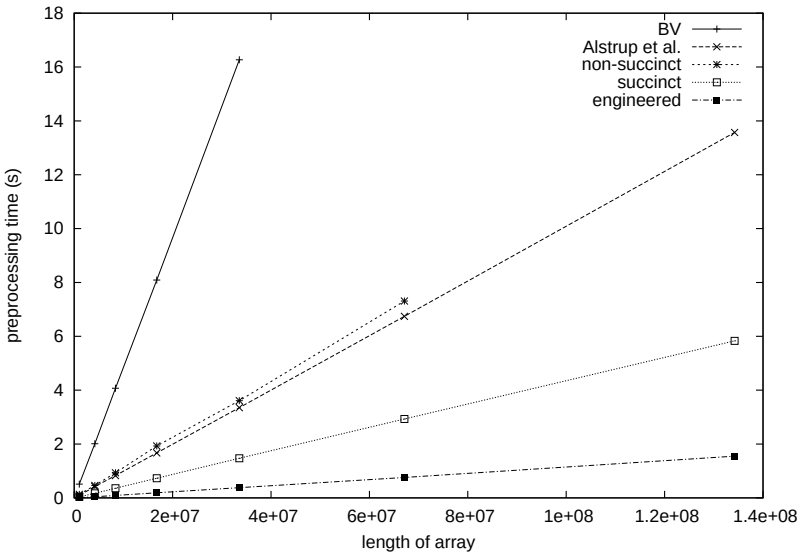


Figure 3.8: Preprocessing times for varying array lengths. For BV, time for constructing the Cartesian Tree is included in the graph.

Fig. 3.8 shows the time spent on preprocessing by the different methods. The ranking of the methods is the same as in Fig. 3.7, which confirms the fact that preprocessing is largely dominated by filling certain tables by relatively straight-forward computations. But still, there are two important deviations from the space consumptions. The first is that our non-succinct scheme has preprocessing time very close to that of Alstrup et al. As they both have the same preprocessing for long queries, this shows that computing the block types and performing a "naive" precomputation of the in-block-queries amounts to approximately the same time as computing the bit-vectors from Sect. 3.4.2 for each block. The second deviation from Fig. 3.7 is that for our succinct and engineered implementation, the difference in preprocessing *time* is much higher than that in *space*. This and the fact that the table of precomputed in-block tables is marginally small for $s = 2^3$ shows that most of the time which is not spent on filling tables comes from the computation of the block-types.

The next test was to evaluate the influence of the query length on the query time. For this we took a random array of length $n = 4 \times 10^7$ and measured the time for random queries of increasing length $l = 3^1, 3^2, \ldots, \lfloor \log_3 n \rfloor$. The results can be seen in Fig. 3.9. As expected, the naive method behaves linearly in the query length (note the logarithmic x-axis) and is only competitive for relatively small query lengths, say $l \leq 3^5$. Concerning the other methods, the Berkman-Vishkin scheme is always the slowest. The remaining four methods form basically two groups. The non-succinct version of our scheme together with Alstrup et al.'s scheme, which are both about twice as fast as BV. For short queries, methods succinct and engineered are about as fast as non-succinct and Alstrup et al., and approximately as fast as BV for long queries (but never slower). A final interesting point to note in Fig. 3.9 is that all curves except

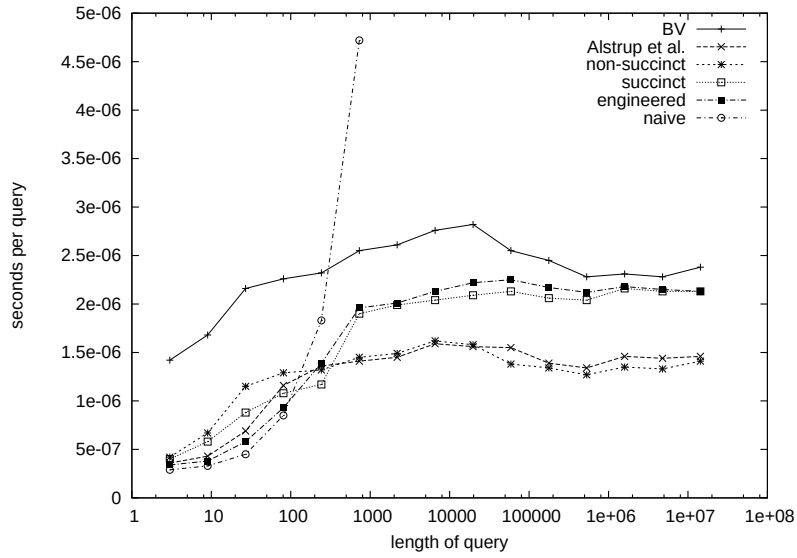
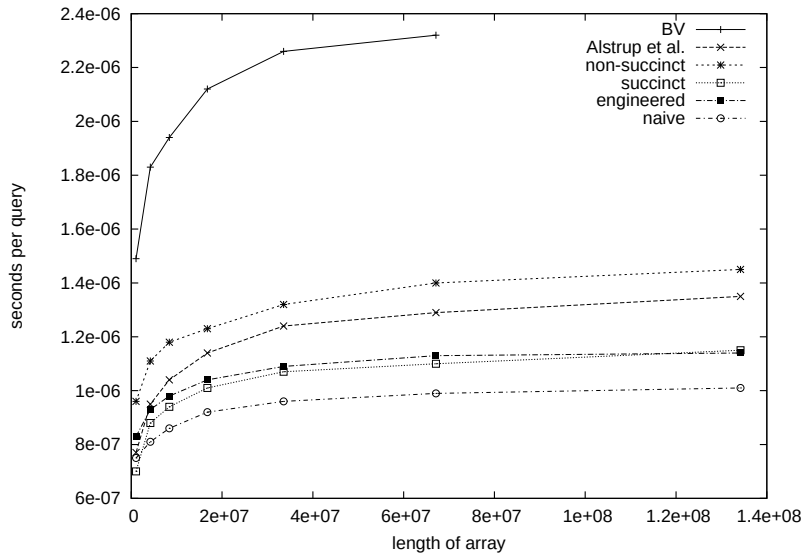


Figure 3.9: Influence of different query lengths on query time (logarithmic x-axis; averaged over 10^6 queries each; without preprocessing times).

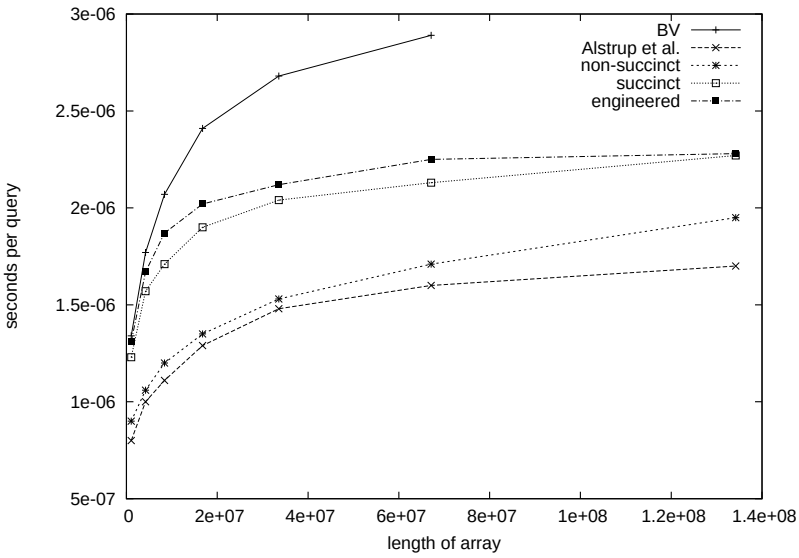
the one for the naive method have their peak at an intermediate query length, and then slightly level off. We can only speculate that this is due to caching phenomena.

In a final test we checked up on the influence of the array length on the query time. We differentiated between constant query lengths (Fig. 3.10) and query lengths that grow with the size of the array (Fig. 3.11). Both tests were further subdivided into short queries (sub-figures (a)) of length 100 or $\log(n)/2$, respectively, and long queries (sub-figures (b)) of length 10,000 or $n/100$, respectively. We do not comment on all graphs separately, but just try to summarize the results in a few points as follows.

- The length of the array has more or less the same influence on all curves.
- The Berkman-Vishkin algorithm is always slowest, compared with any of the other methods.
- The naive method is good for short queries, but we have seen before that it is not competitive for long queries, and therefore excluded it from sub-figures (b).
- For short queries (sub-figures (a)), the differences in query time for the remaining four methods are not as high as for long queries (b).
- For short queries, our non-succinct version is slightly slower than the other three methods, and the engineered version is (marginally) the best for long arrays (apart from the naive method).

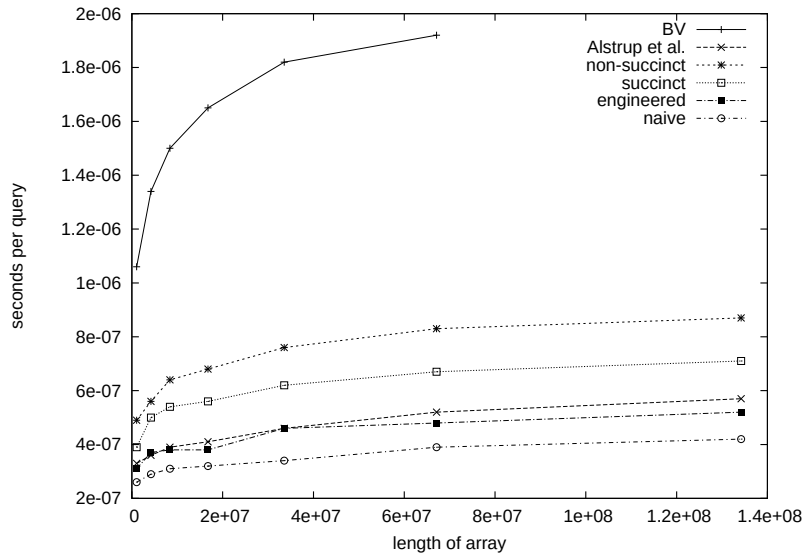


(a) Short queries of length 100.

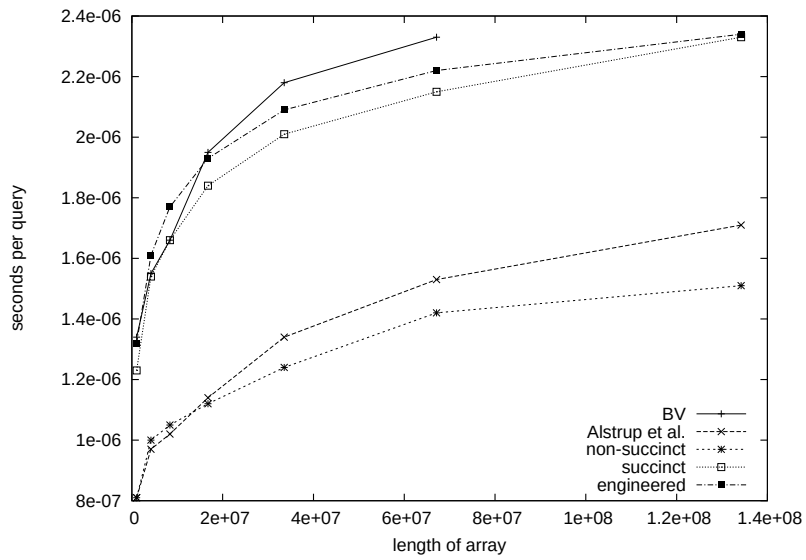


(b) Long queries of length 10,000.

Figure 3.10: Influence of different array lengths on query time (constant query lengths).



(a) Short queries of length $\log(n)/2$.



(b) Long queries of length $n/100$.

Figure 3.11: Influence of different array lengths on query time (growing query lengths).

- For long queries, one can see the same grouping as in Fig. 3.9: succinct and engineered are slower by a factor of ≈ 2 than non-succinct and Alstrup et al.

3.9 How to Further Reduce Space

We thought about further reducing the space of our RMQ-representation. First note that although we have proved the asymptotic optimality of our data structure (Thm. 3.12), this does not mean that a given instance of the problem cannot be represented in less than $2n - o(n)$ bits of space. For example, take the suffix array SA of a string of length n. A lower bound for representing SA can be obtained easily by noting that SA is a permutation of $[1 : n]$, and as all permutations of $[1 : n]$ are indeed a suffix array for some text, we need $\log(n!) = \Theta(n \log n)$ bits in general to represent SA. However, if the size of the alphabet can be upper-bounded by some constant less than n, it has been proved by Schürmann and Stoye (2005) that less bits are sufficient to represent SA. Further, if one takes into account the compressibility of the underlying text, usually measured in the order-k empirical entropy, much better bounds can be shown (cf. Sect. 2.7).

In this section we will explore to which extent some of these ideas can be applied to our problem. The general approach is to try to reduce the space of the type-table T (Sect. 3.5.2.3), as this is where the $2n$ bits come from. All other structures are already of size $o(n)$ bits. We will first give a negative result on the case of small alphabets, but in Sect. 3.9.2 we will see that for compressible input arrays the RMQ-information can be compressed as well.

3.9.1 Small "Alphabets" Don't Help!

The first idea is to see how big table T becomes if the input array consists of numbers from a restricted range $R := [x : y]$. In analogy to the literature on suffix arrays, we call R the *alphabet* of the input array, although we do not usually preprocess *texts* for RMQ. We show in this section that already for very small values of $|R|$, representing T needs asymptotically the same space as for an unrestricted input alphabet.

Recall from Sect. 3.5.2.2 that the number $T(s)$ of blocks of size s that have different RMQs is given by the recurrence

$$T(0) := 1,\ T(s) := \sum_{i=1}^{s} T(i-1)T(s-i) \text{ if } s > 0\ . \tag{3.9}$$

This was derived from the fact that two blocks have the same RMQs iff the minimum occurs at the same position, and the sub-arrays to the left and right of the minimum have the same RMQs. Thus, summing over all possible positions of the minimum as in (3.9) yields the number of different blocks.

Now assume the input array consists of only two different numbers, w.l.o.g. $R = \{0, 1\}$. Then the number of blocks that have different RMQs is given by the recursive formula

$$T_2(0) := 1,\ T_2(s) := \sum_{i=1}^{s} T_2(s-i) \text{ if } s > 0\ , \tag{3.10}$$

as two blocks have the same RMQs iff the first 0 is at the same position i (so all array entries to the left of i are 1's), and the sub-arrays to the right of i have the same RMQs. Unfortunately, the solution to recurrence (3.10) is $T_2(s) = 2^{s-1}$, which can be easily proved by induction. So already for an alphabet of size 2, one needs $\log(2^{s-1}) = s - 1$ bits to describe the type of a block, which is asymptotically only half of the $\log(C_s) = 2s - O(\log s)$ bits for general alphabets.[12]

For general alphabets of size σ, the number of blocks that have different RMQs is given by

$$T_\sigma(0) := 1,\ T_\sigma(s) := \sum_{i=1}^{s} T_{\sigma-1}(i-1)T_\sigma(s-i) \text{ if } s > 0\ , \tag{3.11}$$

as two blocks have the same RMQs iff the minimum occurs at position i, and the sub-arrays to the left and right of i have the same RMQs. As the sub-array to the left of the minimum does not contain the minimal element, it must have an alphabet of size at most $\sigma - 1$, so its number of different blocks is given by $T_{\sigma-1}(i-1)$. Recurrence (3.11) has some very surprising solutions for various values of σ; e.g., we have $T_3(s) = F_{2s-1}$ with F_n being the n'th Fibonacci number, and

$$T_4(s) = \frac{3^{s-1}+1}{2} \text{ for } s > 0\ . \tag{3.12}$$

(Again, both formulas can be proved by induction on s, using some basic identities on the Fibonacci numbers.) We present these formulas only to give an idea of how rapidly $T_\sigma(s)$ converges (with growing σ) to the general $2s - O(\log s)$ bits: already for $\sigma = 4$ we need $\log_2\left((3^{s-1}+1)/2\right) \approx \frac{s-1}{\log_3 2} - 1 \approx 1.58(s-1) - 1$ bits, which is already very close to $2s - O(\log s)$. We conclude from this that small alphabets do not help directly for the encoding of T.

3.9.2 Compression Techniques

Let us now consider input arrays A of length n that are compressible. As we saw in Sect. 2.5, compressibility is usually measured in the order-k entropy $H_k(A)$, as $nH_k(A)$ provides a lower bound on the number of bits needed to encode A by any compressor that considers a *context* of length k when it encodes a symbol in A. We now show that the encoding given by Ferragina and Venturini (2007) is

[12] Ignore for now that for very small values of $|R|$, say $|R| \leq 4$, array T need not be stored at all, as the input array itself can be used for indexing the table of precomputed queries. The aim of this section is solely to explore if bounding the alphabet size helps in general for storing T in less space.

also effective for our type-array T. As usual, let σ denote the size of Σ; i.e., σ is equal to the number of different numbers in A. For completeness of exposition, the encoding (Ferragina and Venturini, 2007) is given as follows (remember that $s = \log n/(2+\delta)$ denotes the block size, $s' = \log^{2+\varepsilon} n$ the superblock size, and B_j the j'th block in A):

- Let $\mathcal{S}$ be the set of occurring block types in A: $\mathcal{S} := \{T[i] : i = 1, \ldots, n/s\}$.
- Sort the elements from $\mathcal{S}$ by decreasing frequency of occurrence in T and let $r(B_j)$ be the rank of block B_j in this ordering.
- Assign to each block B_j a codeword $c(B_j)$ which is the binary string that has rank $r(B_j)$ in $\mathcal{B}$, which is the canonical enumeration of all binary strings: $\mathcal{B} := \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. See Fig. 3.12 for an example (ignore for now the rows labeled V' and V''). The codeword $c(B_j)$ will be used as the type of block B_j; there is *no need to recover the original block types.*
- Build a sequence $V = c(B_1)c(B_2)\ldots c(B_{n/s})$. In other words, V is obtained by concatenating the codewords for each block.
- In order to find the beginning and ending of B_j's codeword in V, we use again a two-level storage scheme (see Munro, 1996) for storing the starting position of B_j's encoding in V: a table D' stores the beginning of the encoding of the superblocks, and table D does the same for the blocks, but storing the position relative to the beginning of the superblock. As the block types are in the range $\{0, \ldots, C_s - 1\}$, the number of bits needed to encode a block is $|c(B_j)| = \log(C_s) = O(\log n)$, so the bit-length of V is at most $|V| = O(\frac{n}{s}\log n) = O(n)$ bits.[13] Consequently, the size of table D' is $|D'| = n/s' \cdot \log|V| = O\left(\frac{n}{\log^{1+\varepsilon} n}\right) = o(n)$, and the size of table D is $|D| = O\left(n/s \cdot \log(s'/s \cdot \log n)\right) = O\left(\frac{n \log\log n}{\log n}\right) = o(n)$ bits. These tables can be filled "on the fly" when writing the compressed string V.

It is obvious how tables D and D' can be used to reconstruct the codeword (and hence the type) of block B_j: simply extract the beginning of block j and that of $j+1$ (if existent); therefore, the above structures *substitute* the type-array T. It goes without saying that table P now needs to be constructed according to the above ranking. The main result of this section can now be stated as follows:

[13] This is just a simple upper bound on $|V|$ that suffices for the moment. Thm. 3.15 gives the real size of V.

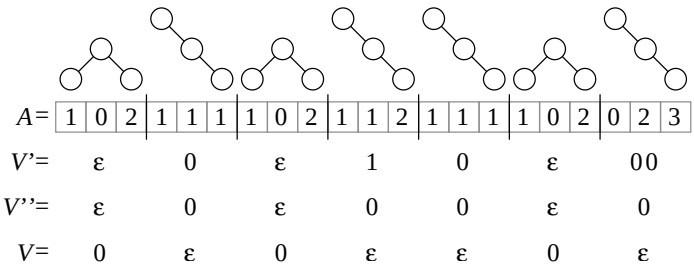


Figure 3.12: Illustration to the compressed representation of RMQ-information. On top of each block of size $s = 3$ one can see the its Canonical Cartesian Tree. The final encoding can be found in the row labeled V; the rows labeled V' and V'' are solely for the proof of Thm. 3.15.

Theorem 3.15 (Entropy-bounds for RMQ)**.** *For an array A with n elements from a totally ordered set of size σ, there exists an algorithm for the RMQ-problem with time complexity $\langle O(n), O(1)\rangle$ and bit-space complexity*

$$\left[\!\left[2n + o(n), nH_k(A) + O\left(\frac{n}{\log n}(k \log \sigma + \log^2 \log n)\right)\right]\!\right] ,$$

simultaneously over all $k \in o(\log n)$.

Proof. Assume first that instead of compressing the block types (i.e., array T), we run the above compression algorithm directly on the contents of A, with the same block size. In other words, we assign to two size-s-blocks the same codeword if they are equal, and these codewords are derived from the frequencies of the blocks in A. See also Fig. 3.12, where the compressed sequence is called V', and $c'(102) = \epsilon$, $c'(111) = 0$, $c'(112) = 1$, and $c'(023) = 00$.

It has been shown (Ferragina and Venturini, 2007, Thm. 3) that the resulting codeword $c'(B_j)$ produced for block B_j is always smaller than if one were to compress the contents of that block with a k-th order Arithmetic Encoder. González and Navarro (2006) have shown that the total output of such an Arithmetic Encoder is bounded by $nH_k(A) + O\left(\frac{nk \log \sigma}{b}\right)$, where b is the block-size (in our case $b = O(\log n)$).

Now observe that if two blocks in the original array A are equal, then they also have the same Cartesian Tree and thus the same block type; so if we encode each block-type with the *shortest* codeword $c'(B_j)$ among all the codewords for blocks that have the same Cartesian Tree, the resulting sequence V'' will always be shorter than V'. See Fig. 3.12 for an example. Now our encoding V cannot be longer than V'', as it assigns even shorter codewords to more frequent types, and therefore obeys the "golden rule of data compression."

Finally, because table M from Sect. 3.5.2.5 is still needed and all other structures are asymptotically smaller, the leading second order term remains $O(n \log^2 \log n / \log n)$. The claim follows. ■

Note that this analysis is quite coarse and certainly "wastes" some space; However, it matches the currently best known results for storing the array A itself in compressed form while still being able to access any $O(\log n)$ contiguous bits in constant time under the RAM model (Sadakane and Grossi, 2006; González and Navarro, 2006; Ferragina and Venturini, 2007). Therefore, even if we proved a better bound on the space of our compressed T, the space needed for storing A itself would be asymptotically larger.

3.9.3 Outlook

There is certainly room for further improvement, especially on the practical side of compression. For example, the fixed and therefore inflexible block division of the algorithm from the previous section is very likely to "destroy" a lot of natural regularity in the input array. A more flexible block division could be obtained by parsing the input array in a Lempel-Ziv manner, thereby dividing it into blocks of varying size whose Cartesian Trees are always "extensions" of Cartesian Trees that have been seen before. Certainly, one would have to make sure that the blocks are never of size $\omega(\log n)$, because otherwise table P would become too large. Also, one should make additional arrangements that blocks with infrequent Cartesian Trees do not become too small, e.g., by merging adjacent small blocks into one block of size $\Theta(\log n)$. With this flexible block-decomposition approach, in order to find the block number of a given index, one would have to introduce another bit array of length n, where a '1' marks the beginning of a block. Then the block-number of a given index could be obtained by a single rank_1-operation. Fortunately, if there are at most $O(n/\log n)$ ones (i.e., blocks), there are solutions for storing the bit-array in $O(n \log\log n/\log n) = o(n)$ bits, while supporting constant-time rank- and select-operations (Raman et al., 2002).

Another open research topic is as follows. We have shown above that RMQ-information can be stored in space which is proportional to the entropy of the *array* which is preprocessed. However, in usual applications this array does not appear from nowhere, as there is usually some underlying structure (e.g., a text) from which the array is generated. Thus the real aim should be to show that it is possible to store RMQ in space bounded by the entropy of the true input. Of course, this would require proprietary solutions for each application. As a concrete example, we have seen that for the highly important problem of finding longest common extensions (Sect. 3.7.2) the RMQs are performed on the LCP-array. Thus it would be desirable to relate the entropy of the underlying text to that of the LCP-array. It is conceivable that this works: *self-repetitions* (see Navarro and Mäkinen, 2007, Def. 9) in the suffix array imply a similar repetition in the LCP-array. The number of self-repetition in the suffix array equals the number of runs in the last column of the Burrows-Wheeler matrix (Burrows and Wheeler, 1994), the latter number being bounded by the text entropy (Navarro and Mäkinen, 2007, Thm. 5). This hints at a possible way for proving a deep connection between text entropy and regularity in LCP-arrays. Such a result would have far-reaching consequences in the whole field

of compressed text indexing; e.g., it would turn Sadakane's compressed suffix tree (2007a) into a fully *compressed* index, in the sense of Navarro and Mäkinen (2007).

3.10 Summary and Discussion

We have presented a direct and easy-to-implement data structure for constant-time RMQ-retrieval that uses $2n+o(n)$ bits of additional space, which is asymptotically optimal and surpasses all previous results on RMQ, both in theory and in practice. The key to our algorithm was the strong connection between Cartesian Trees and RMQs, reflected in the employment of the Catalan- and Ballot numbers. We have also seen how our method leads to space reductions in the computation of lowest common ancestors in binary trees, and to an improved algorithm for the computation of longest common extensions in strings. We further showed that our preprocessing scheme can be compressed by means of a standard entropy-bounded storage scheme.

On the practical side, we have seen that our new scheme has a drastically reduced space consumption, compared to previous approaches. This confirms that our method is not only interesting in theory, but also in practice. However, we have also seen that it is sometimes wiser to spend a little bit less effort in preprocessing, because even for large problem sizes asymptotically slower variants of our algorithm may perform equally fast in practice, while using less space.

CHAPTER

4

Two-Dimensional Range Minimum Queries

4.1 Chapter Introduction

The problem of finding the minimum number in a given range is by no means restricted to one dimension. In this chapter, we investigate the two-dimensional case. Consider an $(m \times n)$-matrix of $N := mn$ numbers. Then one may be interested in preprocessing it so that queries seeking the minimum in a given rectangle can be answered efficiently. This generalization to higher dimensions is quite common in computational geometry, where such "rectangular" queries are typically called *orthogonal range queries* (Agarwal, 2004). An important special case is that of *dominance queries*, where the query rectangle always starts in the origin. The first result on the general 2-dimensional RMQ-problem is due to Gabow et al. (1984), who solve the problem in $O(N \log N)$ preprocessing time and space and $O(\log N)$ query time. Chazelle and Rosenberg (1989) show that with $O(CN)$ preprocessing time and space one can answer queries in $O(\alpha^2(CN, N))$ time for an arbitray C (α being the inverse Ackermann function), and Mäkinen (2003) gives a preprocessing scheme using $O(N \log m)$ preprocessing time and space and $O(1)$ query time. An interesting variant of the dominance query problem, called *RMQ with activation*, has been considered by Abouelhoda and Ohlebusch (2005), who give efficient algorithms for this task.

In this chapter we present a class of algorithms which can solve the 2-dimensional RMQ-problem with $O(kN)$ additional space, $O(1)$ query time, and $O(N(k + \log\log\ldots\log N))$ preprocessing time, where there are $k + 1$ log's, for any $k > 1$. The solution converges towards an algorithm with $O(N \log^* N)$

preprocessing time and space and $O(1)$ query time. Note that k need not necessarily be constant; but if it is, say $k = 2$, then we have an algorithm with $O(N)$ space, $O(N \log\log\log N)$ preprocessing time and $O(1)$ query time. Unlike in the previous chapter, space in this chapter is again measured in words, and not in bits.

While the results presented in this chapter are currently more an academic curiosity, we believe that our solution will turn out to have interesting (theoretical!) applications in the future. One conceivable application comes from computational biology, where one often wishes to identify minimal (or maximal) numbers in a given region of an alignment tableau (Gusfield, 1997). For example, the well-known algorithms from the BLAST-family (Altschul et al., 1990) need to identify *high scoring segments* to speed-up the search for good alignments. A different application where two-dimensional RMQs seem natural are so-called *match chaining problems* (Abouelhoda and Ohlebusch, 2005), where the aim is to chain local optima in order to identify a globally high-scoring chain. Note the similarity of this task to that of finding *maximal scoring subsequences* in one dimension, which, interestingly enough, also has solutions employing (one-dimensional) RMQs, as sketched in Sect. 3.3.5.

4.2 Preliminaries

Let us first give some general definitions. As usual, by $\log n$ we mean the binary logarithm of n, and $\log^{[k]} n$ denotes the k-th iterated logarithm of n, i.e., $\log^{[k]} n = \log\log\ldots\log n$, where there are k log's. Further, $\log^* n$ is the usual *iterated logarithm* of n, i.e., $\log^* n = \min\{k : \log^{[k]} \leq 1\}$.

Now let us formally define the problem which is the issue of this chapter. We are given a 2-dimensional array (which we will often simply call *matrix*) $A[0 : m-1][0 : n-1]$ of size $m \times n$. We wish to preprocess A such that queries asking for the position of the minimal element in an axis-parallel rectangle $[y_1, y_2] \times [x_1, x_2]$, denoted by $\textsc{rmq}(y_1, x_1, y_2, x_2)$, can be answered efficiently. More formally, $\textsc{rmq}(y_1, x_1, y_2, x_2) = \arg\min_{(y,x)\in[y_1:y_2]\times[x_1:x_2]}\{A[y][x]\}$. Throughout this chapter, let $N = mn$ denote the size of the input. As in the previous chapter, there are, of course, trade-offs between preprocessing space and query time; e.g., simply precomputing *all* possible n^2m^2 queries in the input matrix yields $O(N)$ space and constant query time, and doing no precomputations at all and searching the query rectangle naively leads to $O(1)$ space and $O(N)$ query time in the worst case.

4.3 Methods

For a simpler presentation, we assume that the input array is a square, i.e., we have $m = n$ and $N = n^2$. The reader can verify that this assumption is not necessary for the validity of the algorithm. Further, because the query time will be constant throughout this chapter, we do not always explicitly mention

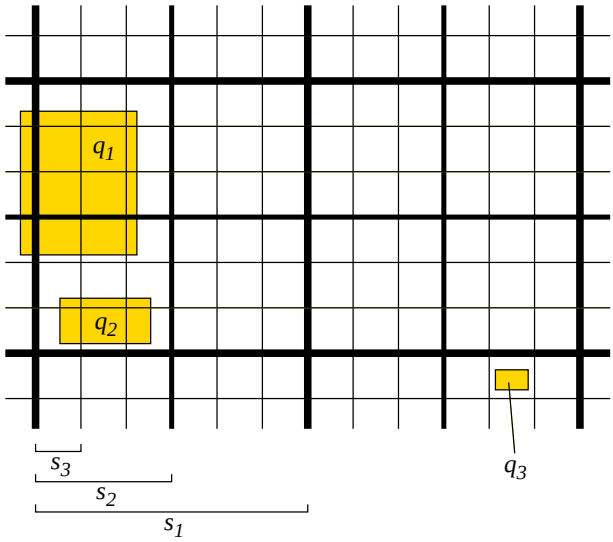


Figure 4.1: Covering the input array with grids of different width. q_1, q_2, q_3 denote queries.

this fact.

We first give a high-level overview of the algorithm (see also Fig. 4.1). The idea is to cover the input array with grids of decreasing widths $s_1, s_2, \ldots,$ thus dividing the array into blocks of decreasing size. For each such grid, we preprocess the array such that queries which *cross* the grid of a certain width s_k, but no grid of width $s_{k'}$ for $k' < k$, can be answered in constant time. Each such preprocessing will use $O(N)$ space and $O(N)$ time to construct. E.g., query q_1 in Fig. 4.1 will be answered on level 1 because it crosses the grid with width s_1, whereas q_2 will be answered on level 3. If the query rectangle does not cross any of the grids (e.g., q_3 in Fig. 4.1), we solve it by having precomputed *all* queries inside such small blocks which we call *microblocks*. If the size of these microblocks is constant, this constitutes no extra (asymptotic) time for preprocessing (leading to the $\log^*$-solution); otherwise we have to employ sorting of the blocks for a constant time preprocessing, leading to the $O(N(k + \log^{[k+1]} N))$ preprocessing time. The details are as follows.

4.3.1 A General Trick for Query Precomputation

Assume we want to answer in $O(1)$ time all queries $\textsc{rmq}(y_1, x_1, y_1 + l_y - 1, x_1 + l_x - 1)$ for y_1 taken from a certain subset of indices $Y \subseteq [0 : n-1]$ (and likewise x_1), and certain query lengths $l_y \in L_y = [1 : |L_y|]$ ($l_x \in L_x$). Then instead of precomputing *all* queries inside the input matrix, it suffices to precompute the answers for query rectangles whose side lengths are a power of 2; i.e., precompute $\textsc{rmq}(y_1, x_1, y_1 + l_y, x_1 + l_x)$ for all $y_1 \in Y, x_1 \in X$, $l_y \in \{2^1, 2^2, 2^3, \ldots, 2^{\lfloor \log |L_y| \rfloor}\}$, and $l_x \in \{2^1, 2^2, 2^3, \ldots, 2^{\lfloor \log |L_x| \rfloor}\}$ and store the results in a table. These precomputations can be done in optimal time using dy-

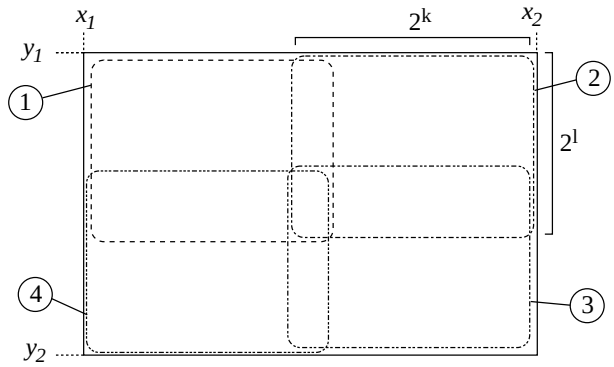


Figure 4.2: Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into four equal-sized overlapping rectangles whose side lengths are a power of two. Taking the position of where the overall minimum occurs is the answer to the query.

namic programming. The reason why precomputing these queries is enough is given by the simple fact that *all* queries can be answered by decomposing them into 4 different rectangles whose side lengths is a power of 2; see Fig. 4.2. Note the similarity to the *sparse table* algorithm for the 1-dimensional solution (Sect. 3.4.1). We denote by $g(|Y|, |X|, |L_y|, |L_x|) := |Y| \times |X| \times \lfloor \log |L_y| \rfloor \times \lfloor \log |L_x| \rfloor$ the space occupied by the resulting table for this kind of preprocessing.

Note how this idea already yields an RMQ-algorithm with $O(N \log^2 n)$ preprocessing time and space and $O(1)$ query time: simply perform the above preprocessing for $X, Y = [0 : n-1]$, $L_x, L_y = [1 : n]$; the space needed is then $g(n, n, n, n) = N \log^2 n$.

4.3.2 Linear Preprocessing of the First Level

We now present a preprocessing to answer all queries which cross the grid for width $s := s_1 := \log n$. The input array is partitioned into blocks of size $s \times s$. Then a query can be decomposed into at most 9 different sub-queries, as seen in Fig. 4.3. Query number 1 exactly spans over at least one block in both x- and y-direction. Queries 2–4 span over at least one block in one direction, but not in the other direction. Queries 6–9 lie completely inside one block (but meet at least one of the four "boundaries" of the block).

Next, we show how to preprocess A such that all queries 1–9 can be answered in constant time. Taking the position where the overall minimum occurs is the final result.

Queries of type 1. We apply the idea from Sect. 4.3.1 on the set $Y = X = L_y = L_x = \{0, s, 2s, \ldots, (n-1)/s\}$; i.e., we precompute $\text{RMQ}(ys, xs, (y + 2^k)s - 1, (x + 2^l)s - 1)$ for all $x, y \in \{0, s, \ldots, (n-1)/s\}$ and all $k, l \in [0 : \lfloor \log(n/s) \rfloor]$. The results are stored in a table of size $g(n/s, n/s, n/s, n/s) = O(n/s \times n/s \times \log n \times \log n) = O(N)$. As usual, queries of this type are then answered by

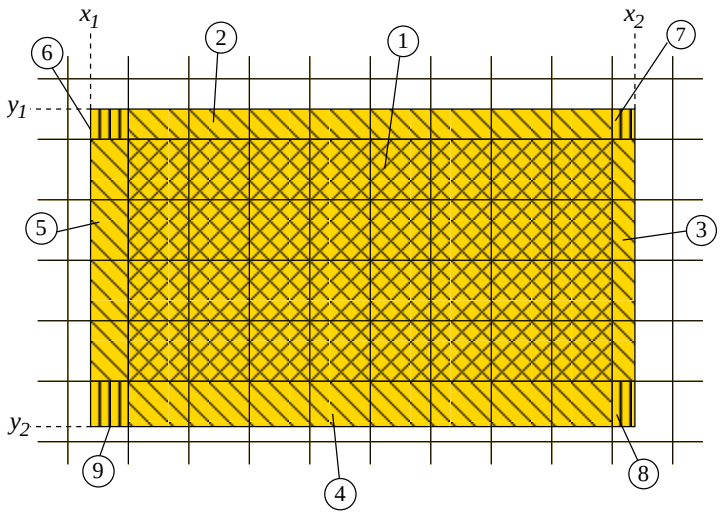


Figure 4.3: Decomposing a query rectangle $[y_1 : y_2] \times [x_1 : x_2]$ into at most 9 sub-queries.

selecting the minimum of at most 4 overlapping precomputed queries.

Queries of type 2–5. We just show how handle queries 2 and 4; the ideas for 3 and 5 are similar. Note that unlike Fig. 4.3 suggests, such queries are not guaranteed to share an upper or lower edge with the grid; the general appearance of these queries can be seen in Fig. 4.4. So the task is to answer all queries $\text{RMQ}(y_1, x_1 s, y_1 + l_y, x_1 s + l_x - 1)$ for all $y_1 \in [0 : n-1]$, $x_1 \in [0 : \lfloor n/s \rfloor]$, $l_y \in [1 : s-1]$ and $l_x \in \{s, 2s, \ldots, \lfloor n/s \rfloor\}$. It is easy to verify that simply applying the trick from Sect. 4.3.1 would result in super-linear space; we therefore have to introduce another "preprocessing layer" by bundling $s' := s^2$ cells into one superblock. Then divide the type 2- or 4-query into one query that spans over several superblocks, and at most two queries that span over several blocks, but *not* over a superblock. All three such queries are handled with the usual idea; this means that the space needed for the superblock-queries is $g(n, n/s', s, n/s') = O(n \times n/\log^2 n \times \log\log n \times \log(n/\log^2 n)) = O(n^2 \log\log n/\log n) = O(N)$. The space for the block-queries is $g(n, n/s, s, s'/s) = O(n \times n/\log n \times \log\log n \times \log\log n) = O(N)$.

Queries of type 6–9. Again, unlike Fig. 4.3 suggests, it is *not* sufficient to precompute queries that have a common border with *two* edges of a block; we also have to precompute queries that share an edge with just *one* block-edge. (E.g., imagine the query in Fig. 4.4 were shifted slightly to the left. Then there would be a part of the query in the block to the very left which only touches the right border of the block.) We just show how to solve queries that share a border with the upper edge of a block (see Fig. 4.5); these structures have to be duplicated for the other three edges. This means that we want to answer $\text{RMQ}(y_1 s, x_1, y_1 s + l_y, x_1 + l_x)$ for all $y_1 \in \{0, \ldots, (n-1)/s\}$, $x_1 \in \{0, \ldots, n-1\}$, and $l_y, l_x \in \{0, \ldots, s-1\}$. In this case, the idea from Sect. 4.3.1 can be applied

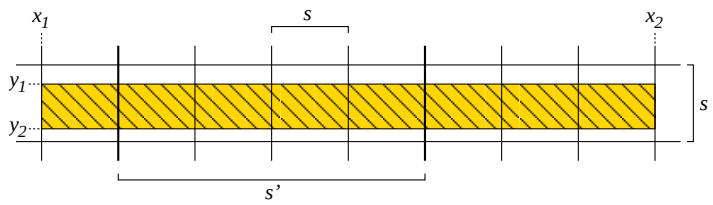


Figure 4.4: Queries spanning over more than one block in x-direction, but not in the y-direction. Sub-queries 2 and 4 from Fig. 4.3 are special cases of these.

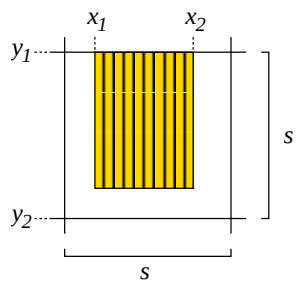


Figure 4.5: Queries lying completely within a block, but sharing the upper edge with it. Sub-queries 8 and 9 from Fig. 4.3 are special cases of these.

directly, leading to a space consumption of $g(n/s, n, s, s) = O(n/\log n \times n \times \log\log n \times \log\log n) = O(N)$.

4.3.3 Recursive Partitioning

We are left with the task to answer RMQs which lie completely inside one of the blocks with side length $s = \log n$. We now offer two recursion strategies which yield the $\log^{[k+1]}$- and $\log^*$-algorithms that have been promised before.

The first idea is to recurse at least one more time into the blocks, and thereafter precomputing all queries which lie completely in one of the microblocks. To be precise, we take each of the $(n/s)^2$ resulting blocks from Sect. 4.3.2 and prepare them with the same method. Then the resulting blocks have side length $s_2 := \log^{[2]} n$. This process can be continued until the resulting blocks have side length $s_k = \log^{[k]} n$ for some $k > 1$. Note that k need not necessarily be constant; all we require is that it be at least 2. As each level needs $O(N)$ space, the resulting space is $O(Nk)$. We now show that already for $k = 2$ we can precompute all queries inside the microblocks in $O(N)$ space and $O(N \log^{[k+1]} N)$ time. We denote by $S := s_k^2$ the size of the microblocks (i.e., the number of elements one microblock contains).

The idea is to precompute all RMQs for all permutations of $[1 : S]$ and look up the result for a certain query block in the right place in this precomputed

table. To do so, assign a *type* to each microblock $[y : y+s_k-1] \times [x : x+s_k-1]$ in A as follows: (conceptually) write the elements from the microblock row-wise into an array $B_{y,x}$; i.e.,

$$B_{y,x}[1,S] = A[y][x : x+s_k-1] \dots A[y+s_k-1][x : x+s_k-1] \ .$$

Then stably-sort $B_{y,x}$ to obtain a permutation π of $\{1,\dots,S\}$ s.th. $B_{y,x}[\pi_1] \leq B_{y,x}[\pi_2] \leq \dots \leq B_{y,x}[\pi_S]$. The index of π in an enumeration of all permutations of $[1 : S]$ is the microblock-type. As there are N/S blocks of size $S = s_k^2$ to be sorted, this takes a total of $O(N/S \times S \log S) = O(N \log^{[k+1]} n)$ time.[1]

The reason for assigning the same type to microblocks whose elements are in the same order can be seen by the following (obvious) lemma:

Lemma 4.1. *Let B_1 and B_2 be two blocks that have the same relative order π as defined above. Then $\text{RMQ}_{B_1}(y_1,x_1,x_2,y_2) = \text{RMQ}_{B_2}(y_1,x_1,x_2,y_2)$ for all values of y_1,x_1,y_2,x_2.* ■

This implies that the following is enough to answer RMQs inside of microblocks: For all permutations π of $\{1,\dots,S\}$, precompute all possible RMQs inside the block

$$\begin{pmatrix} \pi_1 & \dots & \pi_{s_k} \\ \pi_{s_k+1} & \dots & \pi_{2s_k} \\ \vdots & \ddots & \vdots \\ \pi_{(s_k-1)s_k+1} & \dots & \pi_S \end{pmatrix}$$

and store them in a table P (for "precomputed"). As there are s_k^4 possible queries in each of the $S!$ possible microblocks, the size of P is

$$\begin{aligned} s_k^4(S!) &= s_k^4\sqrt{2\pi s_k^2} \cdot \left(\frac{s_k^2}{e}\right)^{s_k^2} \cdot (1+O(s_k^{-2})) \text{ (by Stirling)} \\ &\leq \log^5 \log n \cdot \left(\log^2 \log n\right)^{\log^2 \log n} \cdot (1+O(s_k^{-2})) \text{ (because } k>1) \\ &= (\log \log n)^{2\log^2 \log n+5} \cdot (1+O(s_k^{-2})) \\ &= O(N) \ . \end{aligned}$$

The last equation is true because $b^2 = O(2^b)$, so $(2b^2+5)/\log_b 2 \leq 2^{b+1}$ for large enough b; exponentiating with 2 yields $b^{2b^2+5} \leq 2^{\left(2^{b+1}\right)} = \left(2^{\left(2^b\right)}\right)^2$, which yields the result with $b = \log \log n$. Now to answer a query, simply look up the result in this table.

The second idea is to recurse further into the blocks until the resulting microblocks have constant size; this happens after $O(\log^* n)$ recursive steps. If the resulting micro-blocks have constant size they be sorted in $O(1)$ time each; and because there are $(n/\log^* n)^2$ microblocks this takes a total of $O(N)$ time. The space consumed by this kind of preprocessing is clearly bounded by $O(N \log^* N)$ due to the number of recursive steps performed.

[1] In the special case where the elements from the original array are in the range from 1 to N, we can bucket-sort all block simultaneously in $O(N)$ time.

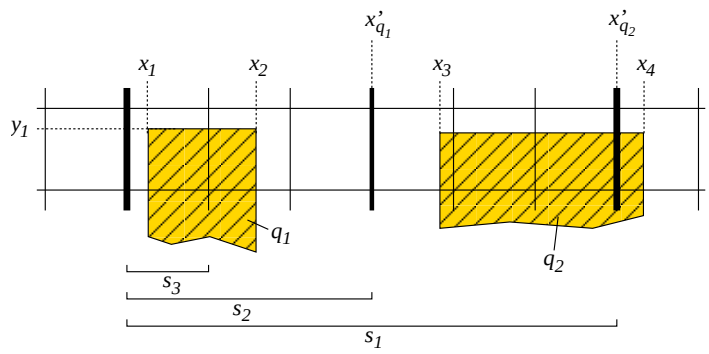


Figure 4.6: How to determine the level on which a specific query has been precomputed.

If we now applied the same recursive steps also for *answering* queries, this would result in $O(k)$ and $O(\log^* N)$ query time, respectively. This is sufficient if k is constant, e.g., with k fixed to 2, this yields an algorithm with $O(N \log\log\log N)$ preprocessing time, $O(N)$ space and $O(1)$ query time. However, if k depends on the input size, and for the $\log^*$-algorithm, this would not yield constant query time. The next section shows how in such cases query time can be reduced to $O(1)$, too.

4.3.4 What's Left: How to Find the Right Grid

For both the $\log^{[k]}$- and the $\log^*$-algorithm it remains to show how to determine in $O(1)$ time the grid with the largest width $s_i = \log^{[i]} n$ such that the query block crosses this grid. In other words, we wish to find the *smallest* $1 \leq i \leq k$ such that the query crosses the grid with width s_i, because at this level the answers have been precomputed and can hence be looked up. We will just show how to do this for the x-direction; the ideas for the y-direction are similar. For simplicity, assume that s_j is a multiple of s_{j+1} for all $1 \leq j < k$.

Let $\textsc{rmq}(y_1, x_1, y_2, x_2)$ be the given query, and let $l_x := x_2 - x_1 + 1$ denote the side length of the query rectangle in x-direction. Let i be defined such that $s_{i-1} > l_x \geq s_i$. This i can be found in $O(1)$ time by precomputing an array $I[1 : n-1]$ with $I[l] = \min\{k : l \geq s_k\}$; then $i = I[l_x]$. Then the query crosses *at least* one column from the s_i-grid, and *at most* one column from the s_{i-1}-grid. As an example, consider query q_1 in Fig. 4.6. We have $s_2 > l_x \geq s_3$, and indeed, q_3 crosses a column from the s_3-grid, and, in this example, no column from the s_2-grid. Likewise, for query q_2 we have $s_1 > l_x \geq s_2$, and it crosses an s_1- and an s_2-column. Let $x' := \lfloor \frac{x_2}{s_{i-1}} \rfloor \cdot s_{i-1}$ be the x-coordinate of where this crossing with a column from the s_{i-1}-grid can occur.

Assume first that $x' \notin [x_1 : x_2]$ (as x'_{q_1} for q_1 in Fig. 4.6). This means that the s_{i-1}-grid does *not* cross the query rectangle in x-direction; and the same must be true for all $i' < i$, as $s_{i'} > s_i$. So we know for sure that i is the smallest

value such that the s_i-grid passes through the query rectangle in x-direction. In this case we are done.

Now assume that $x' \in [x_1 : x_2]$ (as x'_{q_2} for q_2 in Fig. 4.6). In other words, the s_{i-1}-grid crosses the query rectangle in x-direction at position x'. In this case we are not yet done, because it could still be that an $s_{i'}$-column with $i' < i - 1$ also passes through x'. To find the smallest such i', define an array $I'[0 : n-1]$ such that $I'[x] = j$ iff j is the smallest value such that there is a column from the s_j-grid passing through x. This array can certainly be precomputed during the k rounds of the preprocessing algorithm from the previous sections. As an example, in Fig. 4.6 it could still be that a column from a different grid (say from a hypothetical s_0-grid) passes through the query rectangle. But as this *must* happen at x'_{q_2}, we find this value at $I'[x'_{q_2}]$.

In total, we do the above for both the x- and y-direction, and look up the query result at the minimum level i from both steps. If, on the other hand, we find that the query rectangle does not cross a grid in any direction, the result can be looked up in table P.

4.4 Summary and Outlook

We have seen a class of algorithms which solve the two-dimensional RMQ-problem, summarized in the following theorem:

Theorem 4.2 (2-dimensional RMQs)**.** *For any $k > 1$ which may be constant or not, an $(m \times n)$-matrix can be preprocessed in $O(nm(k + \log^{[k+1]}(mn)))$ time such that the position of a minimum value in an axis-parallel query-rectangle can be obtained in constant time, using $O(kmn)$ words of additional space. This converges towards an algorithm with $O(mn \log^*(mn))$ preprocessing time* and *space and $O(1)$ query time.* ■

While some ideas of our algorithm were similar to the one-dimensional counterpart of the problem, others were completely new, e.g., the idea of iterating the algorithm for k levels, while still handling all queries as if they were on the first level, and the idea of sorting the miniblocks in order to derive their block types. Note that preprocessing time of our algorithm is not yet linear in the size of the input, as it is the case for the 1D-RMQ (though being *very* close to linear!). We conjecture that achieving linear time is impossible. In particular, we believe that it should be possible to show that there is no such nice relation as the one between the number of different RMQs and the number of different Cartesian Trees in the one-dimensional case (Lemma 3.9). This could mean that ideas such as sorting or recursing become un-avoidable, thereby hinting at a super-linear lower bound.

A final remark is that our method can be generalized to higher dimensions d using the same techniques as presented in this chapter. However, as with increasing dimension the hyper-rectangular query has to be decomposed into

more and more sub-queries, d will probably appear in both query-time and space.

CHAPTER

5

Improvements in the Enhanced Suffix Array

5.1 Chapter Introduction

Suffix trees (Gusfield, 1997) are a very powerful tool for many tasks in pattern matching (see Apostolico, 1985, for a multitude of their applications). Because of their large space consumption (according to Abouelhoda et al. (2004), up to 20 bytes per input character even with very careful implementations such as the one from Kurtz (1999)), a recent trend in text indexing tries to replace them by adequate array-based data structures. Kasai et al. (2001) showed how algorithms based on a bottom-up traversal of a suffix tree can be simulated by a parallel scan of the suffix- and LCP-array. The Enhanced Suffix Array (ESA) (Abouelhoda et al., 2004) takes this approach one step further by also being capable of simulating top-down traversals of the suffix tree. This, however, requires the addition of another array to the suffix- and LCP-array, the so-called *child-table*. Essentially, the child-table captures the information on how to move from an internal node to its children. This table requires $O(n)$ words (or $O(n \cdot \log n)$ bits). We show in this chapter that the RMQ-information on the LCP-array can be used as an alternative representation of the child-table, thus reducing the space requirement to $O(n/\log n)$ words under the RAM-model (precisely, to $2n + o(n)$ bits). Our representation is not only less space-consuming than that of Abouelhoda et al. (2004), but also much simpler.

One important application of the ESA is that it allows searching an occurrence of a length-m-pattern in the suffix array in $O(m|\Sigma|)$ time. Observe that this is independent of the length n of the indexed text, which stands in high

contrast to the usual search-algorithms in suffix arrays which take $O(m \log n)$ and $O(m + \log n)$ time, respectively (Manber and Myers, 1993). However, this is only appealing if the size of the alphabet is relatively small. We therefore show in Sect. 5.5 how to achieve $O(m \log |\Sigma|)$ searching time by using a variant of the RMQ-preprocessing from Sect. 3.5 that uses only about $2.5n + o(n)$ bits. This variant builds on some nice generalizations of the concepts that we used for deriving the $2n + o(n)$-bit-solution for RMQ. Among other things, we will define a modified Cartesian Tree, which, in turn, involves generalizations of the Catalan and Ballot Numbers.

Some bibliographic remarks are in order. It is well known that with suffix *trees* one can achieve $O(m \log |\Sigma|)$ deterministic searching time if the outgoing edges of each node are implemented as a height-balanced binary tree, e.g., an AVL-tree (Adelson-Velskii and Landis, 1962). On the other hand, if one does not want to store rebalancing-information at each node, randomized binary search trees (Martínez and Roura, 1998) yield the same time bounds in the expected case, while being simpler to implement. Concerning the suffix *array*, Kim et al. (2004) have shown how to achieve $O(m \log |\Sigma|)$ deterministic searching time in the ESA by using a different version of the child-table. This method needs $O(n \log n)$ bits of additional space, but Kim and Park (2005) also gave a succinct version of this child-table using $5n + o(n)$ bits of space. As mentioned above, we reduce this to $\approx 2.5n + o(n)$ bits of space. An additional advantage is that as in the case of the RMQ-based representation of the child-table, our preprocessing scheme for the $O(m \log |\Sigma|)$-search is also much simpler those from Kim et al. (2004) and Kim and Park (2005).

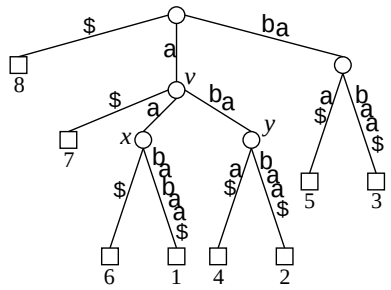
5.2 Enhanced Suffix Arrays

Let T be a text of length n. In its simplest form, the ESA consists of the suffix- and LCP-array for T. The basic idea of the ESA is that internal nodes of the suffix tree correspond to certain intervals (so-called ℓ-intervals) in the LCP-array (recall the definition of SA and LCP in Sect. 2.4):

Proposition 5.1 (Abouelhoda et al., 2004)**.** *Let S, SA, LCP be T's suffix tree, suffix- and LCP-array, respectively. Then the following is equivalent:*

1. *There is an internal node in S representing a sub-word ϕ of T.*
2. *There exist $1 \leq l < r \leq n$ such that*
   a) $\mathsf{LCP}[l] < |\phi|$ *and* $\mathsf{LCP}[r+1] < |\phi|$,
   b) $\mathsf{LCP}[i] \geq |\phi|$ *and* $\phi = T_{\mathsf{SA}[i]..\mathsf{SA}[i]+|\phi|-1}$ *for all* $l < i \leq r$,
   c) $\exists\, q \in \{l+1, \ldots, r\}$ *with* $\mathsf{LCP}[q] = |\phi|$. ■

Parts (a) and (b) say that $[l : r]$ is a maximal interval in SA where all suffixes have a common prefix (namely ϕ), and (c) says that at least two of the suffixes in this interval differ after position $|\phi|$. The pair of indices satisfying point 2



	1	2	3	4	5	6	7	8
SA=	8	7	6	1	4	2	5	3
LCP=	−1	0	1	2	1	3	0	2

Figure 5.1: The suffix tree (top) and the suffix- and LCP-array (bottom) for string T = aababaa\$. The interval $[2:6]$ in LCP is underlined; it corresponds to node v in the suffix tree. Its child-intervals are also underlined; they correspond to the leaf labeled '7' and to the internal nodes labeled x and y, respectively.

of the above theorem are said to form a $|\phi|$-interval $[l:r]$ (also denoted as $|\phi|$-$[l:r]$), and each position q satisfying (c) is called a $|\phi|$-index. For example, in the tree in Fig. 5.1, node v corresponds to the 1-interval $[2:6]$ in LCP and has 1-indices 3 and 5.

We often say that strings corresponding to an internal node in the suffix tree (or, equivalently, to an LCP-interval) are *maximally repeated*[1], because they occur more than once in T, say x times, but all extensions of ϕ (i.e., strings of which ϕ is a proper prefix) occur less than x times.

Let ℓ-$[l:r]$ be any such interval, corresponding to node v in the suffix tree S. Then if there exists an $\ell' > \ell$ such that there is an ℓ'-interval $[l':r']$ contained in $[l:r]$, and no super-interval of $[l':r']$ has this property, then ℓ'-$[l':r']$ corresponds to an internal child of v in S. E.g., in Fig. 5.1, the two child-intervals of 1-$[2:6]$ representing internal nodes in S are 2-$[3:4]$ and 3-$[5:6]$, corresponding to nodes x and y. The connection between ℓ-indices and child-intervals is as follows (Abouelhoda et al., 2004, Lemma 6.1):

Proposition 5.2. *Let $[l:r]$ be an ℓ-interval. If $i_1 < i_2 < \cdots < i_k$ are the ℓ-indices in ascending order, then the child intervals of $[l:r]$ are $[l:i_1-1], [i_1:i_2], \ldots, [i_k:r]$. (Singleton intervals are leaves!)* ■

With the help of this proposition it is possible to simulate top-down traversals of the suffix tree: start with the interval 0-$[1:n]$ (representing the root), and for each interval recursively calculate its child-intervals by enumerating their ℓ-indices. The child-intervals can then be visited in either a depth-first or a breadth-first manner. To find the ℓ-indices in constant time, Abouelhoda et al.

[1] Kasai et al. (2001) use the term "branching."

(2004) introduce a new array $C[1, n]$, the so-called child-table. As the definition of C and the techniques for retrieving the ℓ-indices are relatively technical, we just give an informal description of how this works: first use some special entries in C (called "up" and "down") to locate the leftmost ℓ-index. All following ℓ-indexes are obtained from the other entries in C (called "next"). The next section shows a simpler and more space efficient alternative to the child-table.

5.3 An RMQ-based Representation of the Child-Table

We now show that the child-table is not needed if the LCP-array is preprocessed for constant-time range minimum queries. The following lemma is the key to our new technique:

Lemma 5.3 (Retrieving ℓ-indices via RMQ)**.** *Let $[l : r]$ be an ℓ-interval. Then its ℓ-indices can be obtained in ascending order by $i_1 = \text{RMQ}_{\mathsf{LCP}}(l + 1, r)$, $i_2 = \text{RMQ}_{\mathsf{LCP}}(i_1 + 1, r), \ldots,$ as long as $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(i_k + 1, r)] = \ell$.*

Proof. Because of point 2(b) in Prop. 5.1, the LCP-values in $[l + 1 : r]$ cannot be less than ℓ. Thus any position in this interval with a minimal LCP-value must be an ℓ-index of $[l : r]$. On the other hand, if $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(i_k + 1, r)] > \ell$ for some k, then there cannot be another ℓ-index in $[i_k + 1 : r]$. Because RMQ always yields the position of the *leftmost* minimum if this is not unique, we get the ℓ-indices in ascending order. ■

With Prop. 5.2 this allows us to compute the child-intervals by preprocessing the LCP-array for RMQ. As an example, we can retrieve the 1-indices of 1-$[2 : 6]$ as $i_1 = \text{RMQ}_{\mathsf{LCP}}(3, 6) = 3$ giving interval $[2 : 2]$ (corresponding to leaf 7 in Fig. 5.1), $i_2 = \text{RMQ}_{\mathsf{LCP}}(4, 6) = 5$ giving $[3 : 4]$ (corresponding to node x). Because $\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(6, 6)] = \mathsf{LCP}[6] = 3 > 1$, there are no more 1-indices to be found, so the last child-interval is $[5 : 6]$ (corresponding to y).

We summarize the new result in the following theorem:

Theorem 5.4 (Simulating top-down-traversals of suffix trees with suffix arrays)**.** *Any algorithm based on a top-down traversal of a suffix tree can be replaced by a data structure using $|\mathsf{SA}| + 4n + o(n)$ bits, without affecting its time bounds. Here, $|\mathsf{SA}|$ denotes the space consumed by a data structure which gives constant-time access to each element of SA (usually $n \log n$ bits).*

Proof. As Prop. 5.2 and Lemma 5.3 show, preparing the LCP-array for RMQ is enough for retrieving the child-intervals of a node. Because of Thm. 3.11, this requires $2n + o(n)$ bits. In Sect. 2.7 we have seen that storing the LCP-array takes $2n + o(n)$ bits of space. With Prop. 5.1, the claim follows. ■

Alg. 5.1 shows how to retrieve the child-interval of a given interval by specifying the next character to be read. To be precise, if $[l : r]$ is the ℓ-interval

Algorithm 5.1: How to find the child-interval of $[l : r]$ that represents all suffixes having $a \in \Sigma$ as their next character.

Input: interval $[l : r]$ with $l < r$ and a character a
Output: child-interval of $[l : r]$ for a or $\emptyset$ if nonexistent

1 **function** `getChild` (l, r, a)
2 $i \leftarrow \text{RMQ}_{\mathsf{LCP}}(l + 1, r)$, $\ell \leftarrow \mathsf{LCP}[i]$
3 **repeat**
4 **if** $T_{\mathsf{SA}[l]+\ell} = a$ **then return** $[l : i - 1]$
5 $l \leftarrow i, i \leftarrow \text{RMQ}_{\mathsf{LCP}}(l + 1, r)$
6 **until** $l = r \vee \mathsf{LCP}[i] > \ell$
7 **if** $T_{\mathsf{SA}[l]+\ell} = a$ **then return** $[l : r]$ **else return** $\emptyset$

representing string ϕ and $a \in \Sigma$, then `getChild`(l, r, a) returns the child-interval $[l' : r']$ that represents $\phi a \in \Sigma^\star$, or $\emptyset$ if ϕa does not occur in T. We do this by stepping through the ℓ-indices in ascending order (lines 3–6), for each ℓ-index checking if the corresponding character is equal to a (line 4). The correctness of the algorithm follows directly from the above discussion, and the time bound is clearly $O(|\Sigma|)$. Observe that this procedure corresponds to finding the correctly labeled outgoing edge of a node in a suffix tree.

5.4 Pattern Matching in $O(m|\Sigma|)$ Time

For a given pattern P of length m, the most common task in pattern matching is to check whether P is a substring of T, and to enumerate all *occ* occurrences of P in additional $O(occ)$ time. As already mentioned, Abouelhoda et al. (2004) have shown how this can be done in time $O(m|\Sigma|)$ and $O(m|\Sigma|+occ)$, respectively, by incorporating the child-table. In the highly important case where the alphabet size is constant this yields optimal time bounds. Note again that with a plain suffix array these time bounds cannot be achieved, as they are $O(m \log n)$ and $O(m \log n + occ)$, respectively.

Alg. 5.2 shows how to locate a pattern P in $O(m|\Sigma|)$ time. It is a straightforward adaption of the pattern matching algorithm given by Abouelhoda et al. (2004, Alg. 6.8). The invariant of the algorithm is that *found* is `true` if $P_{1..c}$ occurs in T and has the interval $[l : r]$ in SA. In each step of the loop, method `getChild`(l, r, a) is used to find the sub-interval of $[l : r]$ that stores the suffixes having $P_{1..c+1}$ as a prefix. This is done by invoking method `getChild` from the previous section, which takes $O(|\Sigma|)$ time. The if-statement in lines 5–6 distinguishes between internal nodes ($l > r$) and leaves ($l = r$). The actual pattern matching is done in line 7 of the algorithm. Because c is increased by at least 1 in each step of the loop, the running time of the whole algorithm is $O(m|\Sigma|)$. Note that for large $|\Sigma| \in \omega(\log n)$ one would actually drop back to normal binary search, so even for large alphabets matching takes no more than $O(m \log n)$ time.

Algorithm 5.2: How to locate a pattern in a text in $O(m|\Sigma|)$ time.

Input: pattern $P = P_{1..m}$ to be found in T
Output: interval of P in SA or negative answer

1 $c \leftarrow 0$, *found* $\leftarrow$ `true`, $l \leftarrow 1$, $r \leftarrow n$
2 **repeat**
3 $\quad [l : r] \leftarrow$ `getChild`(l, r, P_{c+1})
4 $\quad$ **if** $[l : r] = \emptyset$ **then return** "not found"
5 $\quad$ **if** $l = r$ **then** $min \leftarrow m$
6 $\quad\quad$ **else** $min \leftarrow \mathsf{min}\{\mathsf{LCP}[\text{RMQ}_{\mathsf{LCP}}(l+1, r)], m\}$
7 $\quad$ *found* $\leftarrow (P_{c+1..min-1} = T_{\mathsf{SA}[l]+c..\mathsf{SA}[l]+min-1})$
8 $\quad c \leftarrow min$
9 **until** $l = r \vee c = m \vee$ *found* $=$ `false`
10 **if** *found* $=$ `true` **then return** $[l : r]$ **else return** "not found"

To retrieve all *occ* occurrences of P, get P's interval $[l : r]$ in SA with Alg. 5.2, and then return the set of positions $\{\mathsf{SA}[l], \mathsf{SA}[l+1], \ldots, \mathsf{SA}[r]\}$. This takes $O(occ)$ additional time.

Theorem 5.5 (Pattern matching in $O(|P| \cdot |\Sigma|)$ time)**.** *For a text T of length n over an alphabet Σ there is a data structure occupying $2n + o(n)$ bits that, together with the suffix- and LCP-array for T, allows the retrieval of all occ occurrences of a pattern P in T in $O(|P| \cdot |\Sigma| + occ)$ time, for* any *alphabet size $|\Sigma|$. This data structure can be constructed in-place in $O(n)$ time.* ■

5.5 Pattern Matching in $O(m \log |\Sigma|)$ Time

5.5.1 Basic Idea

Look again at the string matching algorithm from the previous section (Alg. 5.2). It is obvious that the crucial part of its running time is the call of function `getChild` in line 3, as without this call the complete loop (lines 2–9) takes $O(m)$ time, and this is already the time to read pattern P. So improving the running time of function `getChild` (Alg. 5.1) would result in an immediate improvement of the string matching algorithm.

The search for the correct child-interval in Alg. 5.1 is done by a linear scan of the ℓ-indices in ascending order. Recall that the ℓ-indices are the *minima* in the corresponding ℓ-interval. Thus the fact that we perform a linear search is a consequence of our preprocessing scheme for RMQ, as we have defined RMQ to return the *leftmost* position of a minimum value if this is not unique. Now imagine that we have preprocessed the LCP-array for RMQs such that an RMQ always returns the *median* of all positions where LCP attains the minimum in the query-interval. This would allow us to perform a *binary* search on the ℓ-indices for the correct character, as shown in Alg. 5.3 (assume for now that RMQ^{med} returns the perfect median). Thus, with such a preprocessing for RMQ,

Algorithm 5.3: Locating a child-interval with a "binary" search. The "normal" RMQ in line 5 which returns the leftmost minimum is still necessary to find the right end of the final child-interval.

Input: interval $[l : r]$ with $l < r$ and a character a
Output: child-interval of $[l : r]$ for a or $\emptyset$ if nonexistent

```
1 function getChildBinary (l, r, a)
2 i ← RMQ^med_LCP(l + 1, r)                    {get (pseudo-)median of ℓ-indices}
3 ℓ ← LCP[i]
4 repeat
5     if T_SA[i]+ℓ = a then return [i : RMQ_LCP(i + 1, r) − 1]
6     if T_SA[i]+ℓ < a then l ← i else r ← i − 1        {narrow search interval}
7     i ← RMQ^med_LCP(l + 1, r)                        {get next (pseudo-)median}
8 until l = r ∨ LCP[i] > ℓ
9 if T_SA[l]+ℓ = a then return [l : r] else return ∅
```

the running time could be reduced to $O(\log |\Sigma|)$, thereby improving the overall string matching algorithm to $O(m \log |\Sigma|)$. Throughout this section, assume w.l.o.g. that $|\Sigma| = o(n)$, for otherwise a normal binary search (of the complete interval!) already yields an $O(m \log n) = O(m \log |\Sigma|)$ time solution.

The drawback is that the "range median of minima"-problem is anything but trivial to solve. For this reason, we pursue a slightly less ambitious strategy in this section, namely that of finding a *pseudo-median*. A pseudo-median of minima is a minimum with rank between $\frac{1}{C}y$ and $(1-\frac{1}{C})y$ among the y minima in the query-interval (for some constant C).[2] In our case, it will turn out that $C = 1/16$. We already state here that this leads to the following

Lemma 5.6. *If* RMQ^{med} *returns the position of a minimum in a query interval with rank between* $\frac{1}{16}$ *and* $\frac{15}{16}$ *among all minima in that interval, the number of character comparisons made by Alg. 5.3 is* $\log_{\frac{16}{15}} |\Sigma| \approx 10.7401 \log |\Sigma|$ *in the worst case.* ■

5.5.2 A Pseudo-Median Algorithm for RMQ on LCP

We will now show how to preprocess the LCP-array in linear time such that for a given query-interval $[l : r]$ containing y minima, the returned position has rank between $\frac{1}{16}y$ and $\frac{15}{16}y$ among all positions that attain the minimum value. Such queries are called $\text{RMQ}^{\text{med}}(l, r)$. Our general idea is similar to the RMQ-algorithm in Sect. 3.5: we divide the query interval into at most five sub-intervals, one covering super-blocks, two covering blocks to the left and right

[2] Sadakane (2007a) pursues a similar strategy but does not give any details on how to change the RMQ-precomputation such that it returns a pseudo-median. Thus our description in Sect. 5.5.2.1 also closes the gaps in the above mentioned article. However, our ideas from Sect. 5.5.2.2 are *not* necessary for the compressed suffix tree, for reasons that will become clear in that section. We further mention that the constant 1/12 in (Sadakane, 2007a, Lemma 5) should rather be 1/16, as will become evident from our description.

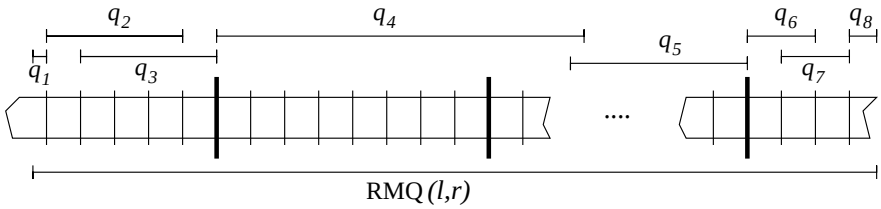


Figure 5.2: The internal decomposition of $\text{RMQ}(l, r)$ into 8 sub-queries. Thin lines denote block-boundaries, thick lines superblock-boundaries.

of the super-blocks, and two in-block queries. There are two major deviations from our previous approach. First, instead of storing the leftmost minimum, each precomputed query stores the *perfect* median among all its minima. And second, in addition to returning the position of the minimum, each sub-query must also return the number of minima in the query-interval.

Recall that the superblock- and block-queries are only precomputed for lengths being a power of two, as every query can be decomposed into two overlapping sub-queries of length 2^l for some l. Thus, a general range minimum query is internally answered by decomposing it in fact into 8 (usually overlapping) sub-queries (see also Fig. 5.2): 2 in-block-queries q_1 and q_8 at the very ends of the query-interval, 4 block queries q_2, q_3, q_6, q_7, and 2 superblock-queries q_4 and q_5. Assume now that for each of these sub-queries q_i we know that at position p_i there is the median among the y_i minima in the respective query interval. Let k be the minimum value in the complete query-interval ($k := \min_{i\in[1:8]}\{\mathsf{LCP}[p_i]\}$), and $I \subseteq [1:8]$ be the set indicating which sub-query-intervals contain the minimum ($I := \{i \in [1:8] : \mathsf{LCP}[p_i] = k\}$. Further, let j be an interval that contains most of these minima, i.e., $j := \arg\max_{i\in I} y_i$. If we then return the value p_j as the final answer to our RMQ, this guarantees that p_j has rank between $\frac{1}{16}y$ and $\frac{15}{16}y$ among all $y := \sum_{i\in I} y_i$ positions that attain the minimum value. To see why this is so, observe that the worst-case scenario is when all 8 sub-query-intervals contain the same number of k's ($I = [1:8]$ and $y_i = y/8 =: y'$ for all i), and the leftmost median p_1 is returned. Then p_1 has rank $\frac{y'/2}{8y'}y = \frac{1}{16}y$ among all minima. The case where the rightmost median p_8 is returned is symmetric.

We thus conclude that if a query interval contains y minima, we have an algorithm which returns a pseudo-median with rank between $\frac{1}{16}y$ and $\frac{15}{16}y$, provided that the precomputed queries return the true median of minima. We now explain how to change the preprocessing from Sect. 3.5 such that this goal is met. We already note here that the block-size is now $s := \log n/(2\log\rho)$, where $\rho := 3 + 2\sqrt{2}$, while the superblock-size remains $s' := \log^{2+\epsilon} n$. The choice of s will become evident in Sect. 5.5.2.2.

5.5.2.1 Preprocessing for Long Queries

Let us first consider the out-of-superblock-queries. We want to temporary preprocess each superblock such that the position of the i'th minimum can be found

in constant time, for all $1 \leq i \leq |\Sigma|$. (Remember that there can be at most $|\Sigma|$ minima in an arbitrary interval in LCP.) We offer two different solution strategies for this tasks, depending on the size of Σ. First, assume the alphabet is relatively small, namely $|\Sigma| = o(\log^{2+\epsilon} n / \log\log n)$. Then for each of the n/s' superblocks, define a temporary array $D'_j[1, |\Sigma|]$ that stores the positions of the minima in superblock B'_j. The D'_j's can certainly be filled in $O(n)$ overall time and need $O(n/s' \times |\Sigma| \cdot \log s') = o(n)$ bits of total space. The i'th minimum can then be obtained by $D_j[i]$. If, on the other hand, $|\Sigma| \neq o(\log^{2+\epsilon} n / \log\log n)$, define D'_j as a bit-vector of length s', where 1s mark the positions of the minima in B'_j. These bit-vectors take a total of $n/s' \cdot s' = n$ bits. Prepare each D'_j for constant time select_1-operations; this takes additional $o(n)$ bits using standard structures for rank/select (see Sect. 2.6). In this case the i'th minimum in B'_j can be obtained by $\text{select}_1(D'_j, i)$.

In addition, we store the actual number of minima in each superblock in the 0th row of a two-dimensional array $Y'[1, n/s'][0, \lfloor \log(n/s') \rfloor]$; i.e., $Y'[i][0]$ equals the number of minima in B'_i. We now show how with the help of these arrays one can compute the two-dimensional table $M'[1, n/s'][0, \log(n/s')]$ that stores the perfect median for out-of-superblock-queries; i.e., $M[i][j]$ stores the median position of all minima in $\mathsf{LCP}[(i-1)s' + 1, (i + 2^j - 1)s']$.

With the D'_j's at hand we can initialize the 0th row of M' with the position of the true median in the superblocks: $M'[i][0] = D'_i[\lfloor Y'[i][0]/2 \rfloor]$. Now suppose we want to fill entry i of table M' on level $j > 0$, i.e., we wish to compute the value $M'[i][j]$ as the median-position of all minima in $\mathsf{LCP}[(i-1)s'+1, (i+2^j-1)s']$ by splitting the interval into two smaller intervals of size $2^{j-1}s'$. By the induction hypothesis, we know that there are $y_l := Y'[i][j-1]$ minima in the left half $\mathsf{LCP}[(i-1)s' + 1, (i + 2^{j-1} - 1)s']$, and $y_r := Y'[i + 2^{j-1}][j-1]$ minima in the right half $\mathsf{LCP}[(i + 2^{j-1} - 1)s' + 1, (i + 2^j - 1)s']$. If the overall minimum occurs only in one half, say the left one, we can safely set $M[i][j]$ to $M[i][j-1]$, and $Y'[i][j]$ to $Y'[i][j-1]$.

On the other hand, suppose that the minimum occurs in both halves. Note that it is all but clear how to find the new median. Fortunately, we have more than $O(1)$ time to calculate it. The true median has rank $r := \lfloor (y_l + y_r)/2 \rfloor$ among all minima in the interval. If $y_l \geq y_r - 1$, we know that this median must be in the left half, and that it must have rank r in there. If, on the other hand, $y_l + 1 < y_r$, the median must be in the right half, and must have rank $r' := r - y_l$ in there. In either case, we recurse in this manner upwards until we reach level 0, where we can select the appropriate minimum from our D'_j-arrays. Due to the "height" of table M', the number of recursive steps is bounded by $O(\log(n/s'))$. In total, filling M' takes $O(n/s' \log(n/s') \log(n/s')) = O(n)$ time. The value $Y'[i][j]$ is simply set to $y_l + y_r$. The size of M' is $o(n)$ bits (see Sect. 3.5.2.5), and table Y' needs $o(n)$ bits, as it stores numbers not bigger than the ones in M'.

The temporary D'_j's (and possibly their additional select-structures) can be deleted once table M' has been filled.

We perform the same preprocessing for the out-of-block queries; i.e., we pre-

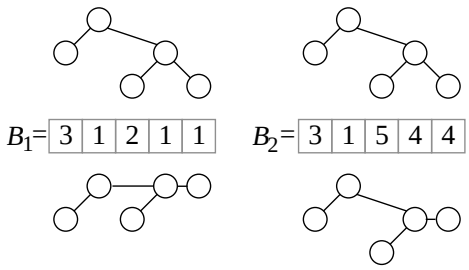


Figure 5.3: Two blocks B_1 and B_2 having the same Canonical Cartesian Tree (top), but a different layout of minima. The Super-Cartesian Trees (bottom) reflect the positions of minima by horizontal edges.

compute a table $M[1, n/s][0, \lfloor \log(s'/s) \rfloor]$ that stores the perfect median of minima for queries that span over blocks, but *not* over a superblock. In this case, if we fill table M superblock-wise, we can always use the first idea from the beginning of this section for finding the i'th minimum in a block: Table $D_j[1, s]$ (storing the absolute positions of minima in block B_j) is only kept in memory for the blocks inside *one* superblock; when M has been filled for this superblock and we move to the next one, the D_j's can be overwritten. Thus, the intermediate space for the D_j's is bounded by $O(s'/s \times s \cdot \log s) = o(n)$, and of course, the total time for computing the D_j's is $O(n)$. The time for filling M with the method from the previous paragraph is $O(n/s \log(s'/s) \log(s'/s)) = O(n \log^2 \log n / \log n) = o(n)$, and the size of M is again $o(n)$ bits (see again Sect. 3.5.2.5). Finally, a new table Y (similar to Y' from the previous paragraph) stores the number of minima inside a precomputed out-of-block query. It needs $O(n/s \times \log(s'/s) \cdot \log s) = o(n)$ bits.

5.5.2.2 Preprocessing for Short Queries

We are still left with the task to precompute the perfect median of the minima inside the blocks of size s. The problem is that we cannot blindly adopt the solution from Sect. 3.5.2.2, because there we regarded blocks with the same Canonical Cartesian Tree as equal, totally ignoring their distribution of minima. As an example, look at the two blocks B_1 and B_2 in Fig. 5.3, where $\mathcal{C}^{\text{can}}(B_1) = \mathcal{C}^{\text{can}}(B_2)$. But for B_1 we want $\text{RMQ}^{\text{med}}(1, 5)$ to return position 4 (the median position of the 3 minima), whereas for B_2 the same RMQ should return position 2, because this is the unique position of the minimum.

We overcome this problem by introducing a new kind of Cartesian Tree which is tailored to meet our special needs for this task:

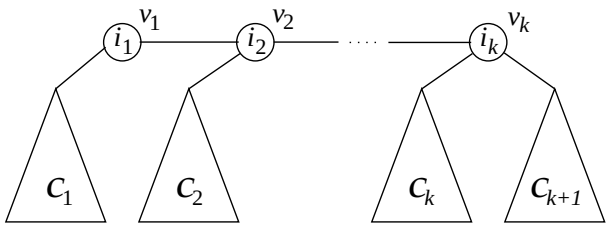


Figure 5.4: Illustration to the definition of Super-Cartesian Trees. The horizontal edges can be considered as right edges with a different "label;" in this sense, the Super-Cartesian Tree is in fact a Schröder Tree.

Definition 5.7. *Let $A[l, r]$ be an array that attains its minima at positions $i_1, i_2, \ldots, i_k$ for some $k \geq 1$. Then the* Super-Cartesian Tree $\mathcal{C}^{\text{sup}}(A)$ *of A is a binary tree recursively constructed as follows:*

- *If $l > r$, $\mathcal{C}^{\text{sup}}(A)$ is the empty tree.*
- *Otherwise, create k nodes $v_1, \ldots, v_k$, where v_j is labeled with i_j. v_1 is the root of $\mathcal{C}^{\text{sup}}(A)$, and v_{i-1} is connected to v_i with a* horizontal *edge for $i > 1$. Recursively construct $\mathcal{C}_1 := \mathcal{C}^{\text{sup}}(A[l, i_1-1])$, $\mathcal{C}_2 := \mathcal{C}^{\text{sup}}(A[i_1+1, i_2-1])$, $\ldots$, $\mathcal{C}_{k+1} := \mathcal{C}^{\text{sup}}(A[i_k + 1, r])$. For $1 \leq i < k$, the left child of v_i is the root of $\mathcal{C}_i$. Finally, the left and right children of v_k are the roots of $\mathcal{C}_k$ and $\mathcal{C}_{k+1}$, respectively. See also Fig. 5.4.*

Note that C^{sup} is in fact a binary tree, where right edges may either be "horizontal" or "vertical." Fig. 5.3 shows C^{sup} for our two example arrays. The reason for calling this tree the *Super-Cartesian Tree* will become clear when we analyze the number of such trees. But let us first give an algorithm to construct $\mathcal{C}^{\text{sup}}$. This is a straight-forward extension of the algorithm for constructing the Canonical Cartesian Tree, treating the "equal"-case in a special manner. So let $\mathcal{C}_i^{\text{sup}}(A)$ be the Super-Cartesian Tree for $A[1, i]$. Then $\mathcal{C}_{i+1}^{\text{sup}}(A)$ is obtained by first climbing up from the rightmost leaf of $\mathcal{C}_i^{\text{sup}}(A)$ towards the root until one finds the first node v_m with label l_m such that $A[l_m] \leq A[i + 1]$, and then inserting a new node w with label $i + 1$ at the correct position. Precisely, let $v_1, \ldots, v_k$ be the nodes on the rightmost path in $\mathcal{C}_i^{\text{sup}}(A)$ with labels $l_1, \ldots, l_k$, respectively, where v_1 is the root and v_k is the rightmost leaf. Let m be defined such that $A[l_m] \leq A[i + 1]$ and $A[l_{m'}] > A[i + 1]$ for all $m < m' \leq k$. Now, if $A[l_m] = A[i + 1]$, connect the new node w (labeled $i + 1$) with a horizontal edge to v_m, remove v_m's right child v_{m+1} and append it as the left child of w. Otherwise (i.e., $A[l_m] < A[i + 1]$), w becomes the right child of v_m, and v_{m+1} becomes the left child of w. With the same reasoning as in Sect. 3.2 it can be seen that the amortized costs for each step are constant, resulting in an overall linear construction time.

The reason for our definition of the Super-Cartesian Tree is the following

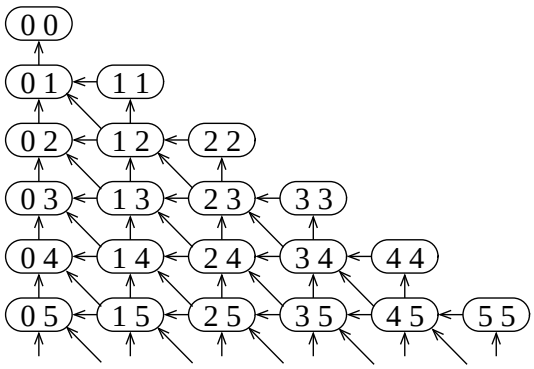


Figure 5.5: An infinite graph whose vertices are $(p\ q)$ for all $0 \leq p \leq q$. There is an edge from $(p\ q)$ to $((p-1)\ q)$ if $p > 0$, and to $(p\ (q-1))$ and $((p-1)\ (q-1))$ if $q > p$.

Lemma 5.8 (Relating RMQs and Super-Cartesian Trees)**.** *Let A and B be two arrays, both of size s. Then $\text{RMQ}_A(i,j) = \text{RMQ}_B(i,j)$ for all $1 \leq i \leq j \leq s$, and the set of positions attaining this minimum is the same in $A[i,j]$ and $B[i,j]$, if and only if $\mathcal{C}^{\text{sup}}(A) = \mathcal{C}^{\text{sup}}(B)$.*

Proof. Similar to the proof of Lemma 3.9, incorporating the changes from Def. 5.7 compared to the Canonical Cartesian Tree. ■

As before, there is a one-to-one correspondence between Super-Cartesian Trees and paths in a certain graph. In this case, the graph is shown in Fig. 5.5. It is an extension of the graph in Fig. 3.3, adding *diagonal* edges from $(p\ q)$ to $((p-1)\ (q-1))$ if $0 \leq p < q$. The bijection is obtained from the above construction algorithm for $\mathcal{C}^{\text{sup}}$, where we first map each step to some number l', which can in turn be mapped to a path in the graph, explained as follows. Let l'_i denote the *height* of the part of the rightmost path that we traverse when constructing $\mathcal{C}^{\text{sup}}_i(A)$ from $\mathcal{C}^{\text{sup}}_{i-1}(A)$. In other words, l'_i counts the number of nodes on the rightmost path that are traversed bottom-up in step i, with the exception that nodes corresponding to equal elements in the array are counted only once, because moving along a horizontal edge in the tree does not change the current height. In the graph in Fig. 5.5, we translate this into a sequence of l'_i upwards moves, and then either a leftwards move if the last comparison was "<," or a diagonal move if it was "=." This gives a one-to-one correspondence between Super-Cartesian Trees and paths from $(s\ s)$ to $(0\ 0)$, because in step i we constrain the number of upwards moves in Fig. 5.5 by the number of strict leftwards moves that have already been made. See the first three columns of Tbl. 5.1 for examples of the correspondence between Super-Cartesian Trees and such paths.

It is well known (Stanley, 1999) that the number of such paths is given by

array A	$\mathcal{C}^{\text{sup}}(A)$	path	number in enumeration
123			0
122			$\hat{C}_{03} = 1$
132			$\hat{C}_{03} + \hat{C}_{02} = 2$
121			$\hat{C}_{03} + \hat{C}_{02} + \hat{C}_{02} = 3$
231			$\hat{C}_{03} + \hat{C}_{02} + \hat{C}_{02} + \hat{C}_{01} = 4$
112			$\hat{C}_{13} = 5$
111			$\hat{C}_{13} + \hat{C}_{02} = 6$
221			$\hat{C}_{13} + \hat{C}_{02} + \hat{C}_{01} = 7$
212			$\hat{C}_{13} + \hat{C}_{12} = 8$
211			$\hat{C}_{13} + \hat{C}_{12} + \hat{C}_{02} = 9$
321			$\hat{C}_{13} + \hat{C}_{12} + \hat{C}_{02} + \hat{C}_{01} = 10$

Table 5.1: Example-arrays of length 3, their Super-Cartesian Trees, and their corresponding paths in the graph in Fig. 5.5. The last column shows how Alg. 5.4 calculates the index of $\mathcal{C}^{\text{sup}}(A)$ in an enumeration of all Super-Cartesian Trees (i.e., Schröder Trees).

the s'th *Super Catalan Number* $\hat{C}_s$ (also known under the name *Little Schröder Numbers* due to their connection with Schröder Trees). These numbers are quite well understood, for our purpose it suffices to know that (see, e.g., Merlini et al., 2004, Thm. 3.6; or Knuth, 1997, exercise 2.2.1–12, where $\hat{C}_s = b_{s+1}/2$)

$$\hat{C}_s = \frac{\rho^s}{\sqrt{\pi s}(2s-1)}(1 + O(s^{-1})) \ , \tag{5.1}$$

with $\rho := 3 + 2\sqrt{2} \approx 5.8284$. As before, this means that we do not have to precompute the in-block-queries for all n/s *occurring* blocks, but only for $O(\rho^s)$ *possible* blocks. (Now our choice for the block-size $s = \log n/(2 \log \rho)$ becomes clear.) The precomputation of the in-block-queries must now be done for all $\hat{C}_s$ possible types; we simply do a naive precomputation of all possible s^2 queries. This table (call it P') needs $O(\hat{C}_s \times s^2 \cdot \log s) = O(\sqrt{n} \log^2 n \log \log n) = o(n)$ bits of space. The time to compute it is $O(\hat{C}_s \times s^3) = O(n)$. (The additional factor s accounts for finding the median of the minima in each step.) As for the long queries, we also have to store the number of minima for each possible query. This table can be filled along with the median-precomputation and can be stored using $o(n)$ bits of space.

5.5.2.3 Computing the Block Types

All that is left now is to assign a *type* to each block that can be used for indexing into table P', i.e., we wish to find a surjection

$$t' : \mathcal{A}_s \to \{0, \ldots, \hat{C}_s - 1\}, \text{ and } t'(B_i) = t'(B_j) \text{ iff } \mathcal{C}^{\text{sup}}(B_i) = \mathcal{C}^{\text{sup}}(B_j) \ , \quad (5.2)$$

where $\mathcal{A}_s$ is the set of arrays of size s. The reason for requiring that arrays with the same Super-Cartesian Tree are mapped to the same type is again given by Lemma 5.8. The easiest way to compute such a bijection would be to actually construct the Super-Cartesian Tree and compute its index in an enumeration of all Schröder Trees. But there is a simpler way, which we will explain in the remainder of this section. Our strategy is to simulate the construction algorithm for Super-Cartesian Trees given before, thereby simulating a walk along the corresponding path in the graph from Fig. 5.5. These paths can be enumerated as follows. First observe that the number of paths from an arbitrary node $(p\ q)$ to $(0\ 0)$ in the graph of Fig. 5.5 is given by the recurrence

$$\hat{C}_{00} = 1, \hat{C}_{pq} = \begin{cases} \hat{C}_{p(q-1)} + \hat{C}_{(p-1)q} + \hat{C}_{(p-1)(q-1)}, & \text{if } 0 \leq p < q \neq 0 \\ \hat{C}_{p(q-1)} & \text{if } p = q \neq 0 \ , \end{cases} \quad (5.3)$$

and $\hat{C}_{pq} = 0$ otherwise. This follows from the fact that the number of paths from a given node to $(0\ 0)$ is given by summing over the number of paths from each cell that can be reached with a single step. The first few such numbers, laid out such that they correspond to the nodes in Fig. 5.5, are

$$\begin{array}{llllll} 1 & & & & & \\ 1 & 1 & & & & \\ 1 & 3 & 3 & & & \\ 1 & 5 & 11 & 11 & & \\ 1 & 7 & 23 & 45 & 45 & \\ 1 & 9 & 39 & 107 & 197 & 197 \ . \end{array} \quad (5.4)$$

Due to this construction the Super Catalan Numbers appear on the rightmost diagonal of (5.4); in symbols, $\hat{C}_s = \hat{C}_{ss}$. Because the numbers $\hat{C}_{pq}$ generalize the Ballot Numbers in the same way as the Super-Catalan Numbers generalize the Catalan Numbers, we christen them *Super-Ballot Numbers*.[3] Although we do not have a closed formula for our Super-Ballot Numbers, we can construct the $s \times s$-array at start-up, by means of (5.3).

We are now ready to describe Alg. 5.4 which computes a function satisfying (5.2). It simulates the construction algorithm for the Super-Cartesian Tree, *without actually constructing it*! The general idea behind the algorithm is to virtually walk along the path from $(s\ s)$ to $(0\ 0)$ in Fig. 5.5, and counting the number of paths that have been skipped by making an upwards (line 6) or diagonal move (line 11). At the beginning of step i of the outer for-loop, the

[3] There is another generalization of Ballot Numbers due to Gessel (1992) that goes under the name Super-Ballot Numbers, which should not be confused with our numbers.

Algorithm 5.4: A modified algorithm to compute the type of a block B_j

Input: a block B_j of size s
Output: the type of B_j, as defined by Eq. (5.2)

```
1  let R be an array of size s + 1                    {simulates rightmost path}
2  R[1] ← −∞                                          {stopper element}
3  q ← s, N ← 0, h ← 0                                {h = #horizontal edges in C^sup_{i−1}(B_j)}
4  for i ← 1, ..., s do
5      while R[q + i + h − s] > B_j[i] do
6          N ← N + Ĉ_{(s−i)q} + Ĉ_{(s−i)(q−1)}        {upwards move}
7          while R[q + i + h − s − 1] = R[q + i + h − s] do h−−
8          q−−
9      endw
10     if R[q + i + h − s] = B_j[i] then
11         N ← N + Ĉ_{(s−i)q}                         {accounts for diagonal move}
12         h++, q−−
13     endif
14     R[q + i + h − s + 1] ← B_j[i]                  {push B_j[i] on R}
15 endfor
16 return N
```

position in the graph is $(s-i+1)\ q$, and h keeps the number of horizontal edges on the rightmost path in $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$. Array R simulates a stack keeping the elements on the rightmost path of $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$, with $q+i+h-s$ pointing to the top of the stack. The loop in line 7 simulates the traversal of horizontal edges in $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$ on the rightmost path by removing all elements equal to the top element on the stack. The if-statement in line 10 accounts for adding a new horizontal edge to $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$, which is translated into a diagonal move in Fig. 5.5. In total, if we denote by l'_i the number iterations of the while-loop from line 5 to 9 in step i of the outer for-loop, l'_i is exactly the height of the part that is removed from the rightmost path. It follows from the previous discussion that Alg. 5.4 correctly computes a function satisfying (5.2) in $O(s)$ time. See the last column of Tbl. 5.1 for examples of the type-calculation.

We store the type of each block in an array $T'[1, n/s]$. Its size is

$$|T'| = \frac{n}{s}\lceil \log \hat{C}_s \rceil \leq \frac{n}{s} \log \rho^s = n \log \rho \approx 2.54311\, n$$

bits. We remark that although `getChildBinary` still needs the "normal" RMQs that return the leftmost minimum, T' can also be used for indexing into the table of precomputed normal RMQs — there is not need to store the type array from Sect. 3.5.2.3 (which would result in another $2n + o(n)$ bits of space)!

5.5.3 Summing Up

The main result of this section is the following

Theorem 5.9 (Pattern matching in $O(|P| + \log|\Sigma|)$ time)**.** *For a text T of length n over an alphabet Σ there is a data structure with space-occupancy of $\approx 2.54311\,n + o(n)$ bits that, together with the suffix array and the LCP-array for T, allows the retrieval of all occ occurrences of a pattern P in T in $O(|P|\log|\Sigma| + occ)$ time, for* any *alphabet size $|\Sigma|$. This data structure can be constructed in $O(n)$ time, and the additional space at construction time is $o(n)$ bits.*

Proof. All that remains to show is the statement on the additional space consumption at construction time. It is clearly $o(n)$ if $|\Sigma| = o(\log^{2+\epsilon} n/\log\log n)$. On the other hand, if $|\Sigma|$ is larger, one simply has to precompute the long queries *before* the short queries. Then the n bits needed for the bit-vectors D'_j can be re-used for the 2.54 bits needed for the type-table T'. ■

The key to this result was a pseudo-median algorithm for RMQ, which led us to a natural generalization of Cartesian Trees, involving generalizations of the Catalan and Ballot Numbers.

We wish to emphasize the fact that our ideas are also compatible with Sadakane's compressed suffix trees (2007a). In this case, our type-table T' from Sect. 5.5.2.2 is not necessary, as the balanced parentheses sequence of the suffix tree already respects the layout of the minima inside the blocks. However, the ideas from Sect. 5.5.2.2 can be transferred one-to-one, and thus close the aforementioned gap in the compressed suffix tree.

A further advantage of our $O(m\log|\Sigma|)$-search is that it is perfectly compatible with compressed representations of suffix arrays (Sect. 2.7). For example, combining the Compressed Suffix Array due to Grossi and Vitter with our search strategy, locating all *occ* occurrences takes $O((m\log|\Sigma| + occ)\log^{\alpha}_{|\Sigma|} n)$ time ($0 < \alpha \leq 1$), for *any* alphabet size $|\Sigma|$, while needing only $\alpha^{-1}H_0 n + O(n)$ bits in total (H_0 being the empirical order-0 entropy of the input text). If *occ* is not too small, this is a significant improvement over the currently fastest locating time in compressed indexes (Ferragina et al., 2007), which takes $O(m + (m + occ\log^{1+\beta} n)\log|\Sigma|/\log\log n)$ time to locate *occ* occurrences ($0 < \beta < 1$), and even this only for $|\Sigma| = o(n/\log\log n)$.

CHAPTER

6

String Mining Problems

6.1 Chapter Introduction

Mining in databases of graphs, trees, and sequences has attracted a lot of interest in recent years, starting with the famous Apriori algorithm for mining frequent itemsets (Agrawal et al., 1993; Agrawal and Srikant, 1994). The typical characteristics of data mining problems is that the patterns to be searched are a priori *unknown*. This means that the user can impose certain conditions on patterns that must be fulfilled to make a pattern be part of the solution. In a certain sense, this is the complete opposite of usual search algorithms such as exact string matching algorithms (cf. Sect. 2.2), where the input is usually the pattern to be found, and the output is the number (and possibly positions) of all matches. In the setting of data mining, the input would be the number of matches, and the output could be all *patterns* that occur at least that often in the data.

In this chapter, we focus on string mining under frequency constraints, i.e., predicates over patterns depending solely on the frequency of their occurrence in the data. This category encompasses combined minimum/maximum support constraints (De Raedt et al., 2002), constraints concerning emerging substrings (Chan et al., 2003), and possibly other constraints concerning statistically significant substrings. We briefly describe the two most common problems in more detail:

- Frequent String Mining: the usual setting here is that we are given two databases, one containing *positive*, the other *negative* patterns. Then one might be interested in extracting all patterns that pass a certain minimum frequency threshold in the positive database, but do not occur too often

in the negative database. The biological relevance of this task can be seen in the following example: suppose a genetic disease is conjectured to be caused by a defect on the X-chromosome, but it is unknown where and how this failure occurs. One can then collect the genetic sequences of the X-chromosome of 1000 ill patients in the positive database, and likewise the genetic sequences of 1000 healthy persons in the negative database. Then all patterns that occur frequently (or always) in the positive database and not too often (or never) in the negative database are potential indicators of the genetic defect under consideration.

- Emerging Substring Mining: this is an extension of the frequent string mining problem and considers patterns as relevant if they have a certain *growth-rate*, defined as the ratio of the relative frequency in the positive database to the relative frequency in the negative database. In this setting, constraints are often easier to formulate, as only one quantity needs to be specified. However, because one is mostly interested in patterns that have a certain statistical significance, one usually specifies an additional constraint which guarantees that the solution patterns have a certain minimal frequency in the positive database.

Further potential application areas of both methods are, among others, finding discriminative features for sequence classification (Birzele and Kramer, 2006), discovering new binding motifs of transcription factors, identifying gene-coding regions in DNA, and microarray design. In the latter example the goal is to find probes (short stretches of sequence spotted on a microarray) differentiating well between groups of sequences. Additionally, the probes have to possess certain physico-chemical properties to qualify them for inclusion on the microarray. Outside the field of computational biology, we mention automatic language classification of texts, spam-recognition of e-mails, and distinction between melodic and non-melodic patterns in MIDI data.

In this chapter, we present an algorithm that is able to answer frequency-based queries optimally, that is, in time linear in the size of the input databases, plus the time to output the solution patterns. The only two assumptions we make is that the number of given databases is constant, and that the frequency-based predicates can be evaluated in constant time. Both assumptions are highly realistic; all of the above mentioned applications can be modeled with just a handful of databases, and in most cases there are only two sets (positive and negative). It is interesting to note that no optimal algorithms are known for other pattern domains such as itemsets or graphs, or there are even hardness results (Wang et al., 2005).

6.2 Formal Problem Definition

We consider patterns from the domain of strings. Extending the notation from Sect. 3.3.2, we will write $\textsc{lce}(\phi, \psi)$ denotes the *longest common extension* of ϕ and ψ, for $\phi, \psi \in \Sigma^\star$. For example, $\textsc{lce}(\texttt{aab}, \texttt{abab}) = \texttt{a}$. Given a set (or

database) $\mathcal{D} \subseteq \Sigma^\star$ with strings over Σ, we write $|\mathcal{D}|$ to denote the number of strings in $\mathcal{D}$, and $\|\mathcal{D}\|$ to denote their total length, i.e., $\|\mathcal{D}\| = \sum_{\phi \in \mathcal{D}} |\phi|$. We define the *frequency* and the *support* of a pattern $\phi \in \Sigma^\star$ in $\mathcal{D}$ as follows:

$$\mathsf{freq}(\phi, \mathcal{D}) := |\{d \in \mathcal{D} \ : \ \phi \trianglelefteq d\}|, \ \mathsf{supp}(\phi, \mathcal{D}) := \frac{\mathsf{freq}(\phi, \mathcal{D})}{|\mathcal{D}|}$$

Note that this is not the same as counting all occurrences of a ϕ in $\mathcal{D}$, because one string in the database could contain multiple occurrences of ϕ.[1] The main contribution of this chapter is to show how one can compute the frequencies (or support) of all strings occurring at least once in one of the databases in optimal time, i.e., in time linear in the size of the input databases (assuming the number of databases is constant). This allows us to solve frequency-related mining queries in optimal time, i.e., in time linear in the sum of the input- and the output-size. Naturally, the query must be computable from the frequency (or support) in constant time.

We now introduce three example problems that can be solved optimally with our approach. The first one is as follows.

Problem 6.1. Given m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ of strings over Σ (constant m) and m pairs of frequency thresholds $(min_1, max_1), \ldots, (min_m, max_m)$, the *Frequent Pattern Mining Problem* is to return all strings $\phi \in \Sigma^\star$ that satisfy $min_i \leq \mathsf{freq}(\phi, \mathcal{D}_i) \leq max_i$ for all $1 \leq i \leq m$. In accordance with the data mining literature (e.g., see Mannila and Toivonen, 1997), this set of solution is often denoted by Th (for "theory").

This problem has been addressed by many authors using different solution strategies and data structures (De Raedt et al., 2002; Fischer and De Raedt, 2004; Lee and De Raedt, 2005; Fischer et al., 2005), but none of these are optimal.

Example 6.1. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\mathcal{D}_1 = \{\mathtt{bbabab}, \mathtt{abacac}, \mathtt{bbaaa}\}, \mathcal{D}_2 = \{\mathtt{aba}, \mathtt{babbc}, \mathtt{cba}\}$, $min_1 = 2$, $max_1 = \infty$, $min_2 = -\infty$, and $max_2 = 2$. Then $Th = \{\mathtt{ab}, \mathtt{aba}, \mathtt{bb}, \mathtt{bba}\}$. Note in particular that because $\mathtt{ba}$ is a substring of all 3 strings in $\mathcal{D}_1$ it satisfies the minimum frequency constraint, but is not part of Th, because its frequency in $\mathcal{D}_2$ is also 3, which is too high.

The size of the solution space Th can be quite big; as a worst case example, assume that we are only given one database $\mathcal{D}_1$, and the thresholds are $min_1 = 1$, $max_1 = \infty$. If $\mathcal{D}_1$ consists of a single string s which is composed of n different letters, then *all* $\Theta(n^2)$ substrings of s are in the solution space, so $\|Th\| = \Theta(n^3)$. This space can be reduced if, instead of enumerating all patterns in Th, one considers a different *representation* of Th, similar to the idea of

[1] Our algorithm can also be used to solve the simpler problem of counting all occurrences of a pattern in the database; for this one only has to calculate the S-counters defined by (6.5) and (6.6), and *not* the C-counters from Sect. 6.3.2.

Gusfield and Stoye (2004) for computing all tandem repeats in a string by returning a suffix tree where all such repeats are marked. In our case, we will see that it is possible to return a "labeled" suffix array from which all solution patterns can be extracted, thereby bounding the size of the output by $O(n)$.

Next, we consider a 2-class problem for a (usually positive) database $\mathcal{D}_1$ and a (usually negative) database $\mathcal{D}_2$. We define the *growth-rate* from $\mathcal{D}_2$ to $\mathcal{D}_1$ of a string ϕ as

$$\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) := \frac{\mathsf{supp}(\phi, \mathcal{D}_1)}{\mathsf{supp}(\phi, \mathcal{D}_2)}, \text{ if } \mathsf{supp}(\phi, \mathcal{D}_2) \neq 0 \ ,$$

and $\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) = \infty$ otherwise. The following definition is motivated by the problem of mining Emerging Patterns (Dong and Li, 1999):

Problem 6.2. Given two databases $\mathcal{D}_1$ and $\mathcal{D}_2$ of strings over Σ, a support threshold ρ_s ($1/|\mathcal{D}_1| \leq \rho_s \leq 1$), and a minimum growth rate $\rho_g > 1$, the *Emerging Substrings Mining Problem* is to find all strings $\phi \in \Sigma^\star$ such that $\mathsf{supp}(\phi, \mathcal{D}_1) \geq \rho_s$ and $\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \geq \rho_g$.

The patterns satisfying both the support- and the growth-rate condition are called *Emerging Substrings* (ESs). ESs with an infinite growth-rate are called *Jumping Emerging Substrings* (JESs), because they are highly discriminative for the two databases. The only known solution for finding ESs (Chan et al., 2003) is quadratic in the input size. The following example will be continued throughout this chapter.

Example 6.2. Let $\mathcal{D}_1 = \{\texttt{aaba}, \texttt{abaaab}\}$, $\mathcal{D}_2 = \{\texttt{bbabb}, \texttt{abba}\}$, $\rho_s = 1$, and $\rho_g = 2$. Then the emerging substrings from $\mathcal{D}_2$ to $\mathcal{D}_1$ are `aa`, `aab` and `aba`. In this case, these are also the JESs.

As a last example problem that can be solved optimally with our method we mention the χ^2-test.

Problem 6.3. Given m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ of strings over Σ and a threshold ρ. Let $n = \sum_{j=1}^{m} |\mathcal{D}_j|$ be the total number of strings, $f = \sum_{i=1}^{m} \mathsf{freq}(\phi, \mathcal{D}_i)$ the total frequency of ϕ, and $\mathbf{E}_j = f \cdot |\mathcal{D}_j|/n$ be the expected value of ϕ's frequency. Then ϕ is significant if it passes the χ^2-test, i.e., if $\chi^2 = \sum_{j=1}^{m} \frac{(\mathsf{freq}(\phi, \mathcal{D}_j) - \mathbf{E}_j)^2}{\mathbf{E}_j} \geq \rho$.

6.3 The New Algorithm

In this section we present our linear-time algorithm for answering frequency-related mining queries (e.g., emerging substrings). Logically, the algorithm can be divided into three main phases: (1) Preprocessing, (2) Labeling, and (3) Extraction. The preprocessing step constructs all necessary data structures: the suffix- and LCP-array, and the preprocessing for RMQ. The labeling step performs one scan over the LCP-array and does the principal work for a fast

calculation of the string-frequencies. Finally, the extraction step scans the LCP-array again, this time simulating a depth-first traversal of the suffix tree, and returns all strings passing the frequency-based criterion.

The main idea for computing the frequencies is due to Hui (1992) and works as follows. Let $\mathcal{D}_j = \{s^{j,1}, \ldots, s^{j,|\mathcal{D}_j|}\}$ be the given databases ($1 \leq j \leq m$). For the strings ϕ occurring in any of the databases, we compute the *total* number of occurrences in $\mathcal{D}_j$ and store the respective numbers in $S_{\mathcal{D}_j}(\phi)$.

$$S_{\mathcal{D}_j}(\phi) = |\{(i,k) : s^{j,k}_{i..i+|\phi|-1} = \phi\}| \tag{6.1}$$

We further compute so-called *correction terms* $C_{\mathcal{D}_j}(\phi)$ that count how often string ϕ has a repetition in the *same* string of $\mathcal{D}_j$:

$$C_{\mathcal{D}_j}(\phi) = \sum_{\substack{s^{j,k} \in \mathcal{D}_j \\ \phi \sqsubseteq s^{j,k}}} (|\{i : s^{j,k}_{i..i+|\phi|-1} = \phi\}| - 1) \tag{6.2}$$

Then $\mathsf{freq}(\phi, \mathcal{D}_j)$ clearly equals $S_{\mathcal{D}_j}(\phi) - C_{\mathcal{D}_j}(\phi)$. In our example, $S_{\mathcal{D}_1}(\mathtt{ab}) = 3$ (there are 3 occurrences of `ab` in $\mathcal{D}_1$) and $C_{\mathcal{D}_1}(\mathtt{ab}) = 1$ (`ab` is repeated once in the second string $s^{1,2} = \mathtt{abaaab}$ of $\mathcal{D}_1$). We will see in Sect. 6.3.3 that it is not very hard to compute the S-numbers in linear time; the real difficulty lies in the computation of the C-numbers. The following lemma suggests how the LCP-array can be used to calculate these correction terms:

Lemma 6.1 (Computation of correction terms)**.** *For any string ϕ occurring in $\mathcal{D}_j$, $C_{\mathcal{D}_j}(\phi)$ is given by the number of times that ϕ is a prefix of the longest common extension of two lexicographically adjacent suffixes from the same string $s^{j,k}$ in $\mathcal{D}_j$:*

$$C_{\mathcal{D}_j}(\phi) = \sum_{k=1}^{|\mathcal{D}_j|} \left| \left\{ 1 \leq i < l : \phi \sqsubseteq \mathrm{LCE}\left(s^{j,k}_{\mathsf{SA}[i]..l}, s^{j,k}_{\mathsf{SA}[i+1]..l}\right) \right\} \right| ,$$

where l denotes the length of $s^{j,k}$, and $\mathsf{SA}[1,l]$ the suffix array for $s^{j,k}$.

Proof. We just prove the claim for a fixed string $s^{j,k}$; the fact that the sum over these numbers equals $C_{\mathcal{D}_j}(\phi)$ follows directly from the sum in (6.2). Let $l := |s^{j,k}|$, and $I := \{i_1, i_2, \ldots, i_x\}$ be the set of all positions in $s^{j,k}$ such that the suffix of $s^{j,k}$ starting at $i \in I$ is prefixed by ϕ, i.e., $\phi \sqsubseteq s^{j,k}_{i..l}$ for all $i \in I$. Because of (6.2), we need to prove that if ϕ occurs in $s^{j,k}$, then $|I| - 1$ equals the number of times that ϕ is a prefix of $\mathrm{LCE}(s^{j,k}_{\mathsf{SA}[i]..l}, s^{j,k}_{\mathsf{SA}[i+1]..l})$. But this is not hard: Due to the lexicographic order of the suffix array, I forms an interval in SA, say $[x,y]$. And because the suffixes $t_{\mathsf{SA}[x]..l}, \ldots, t_{\mathsf{SA}[y]..l}$ are all prefixed by ϕ, the longest common extension of lexicographically adjacent suffixes in I also starts with ϕ, and these are the only such LCEs. The claim follows. ■

As an example, the suffixes of $s^{1,2} = \mathtt{abaaab}$ are (in lexicographic order) `aaab`, `aab`, `ab`, `abaaab`, `b`, and `baaab`. The third and the fourth have `ab` as their

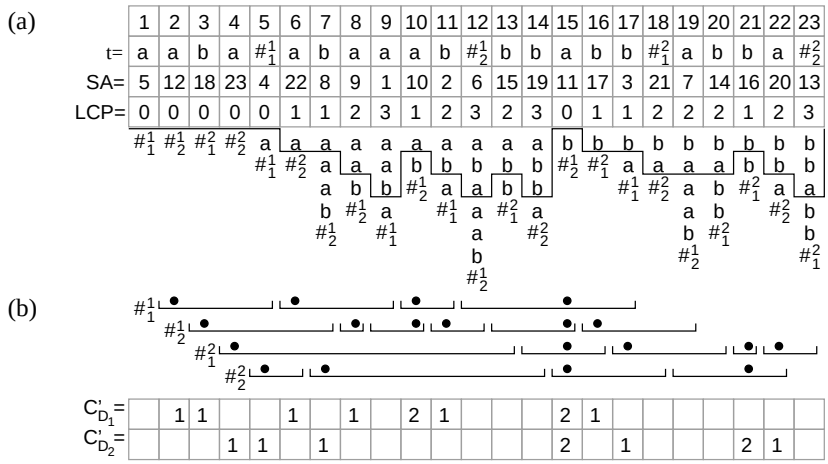


Figure 6.1: (a) The suffix array for w and its LCP-table. Below position i we draw the string $t_{\mathsf{SA}[i]..n}$ until reaching the first end-of-string marker. The solid line going through these strings indicates the LCP-values. (b) Computation of the C'-numbers. The intervals are those for which range minimum queries on LCP are executed; the position of the minimum is depicted by a solid circle. Empty fields in the two arrays denote 0.

longest common extension. Because no suffixes from $s^{1,1} = \texttt{aaba}$ have $\texttt{ab}$ as their longest common extension, the value of $C_{\mathcal{D}_1}(\texttt{ab})$ is 1.

The calculation of the correction terms is done in phase (2) and (3) of our algorithm. In phase (2), we create auxiliary arrays that allow an easy computation of the actual correction terms. The computation of the C-terms is then done along with the computation of the S-numbers in phase (3). The following sections describe the three phases in greater detail.

6.3.1 Preprocessing

We form a (conceptual) string $s^{1,1}\#_1^1 \dots s^{1,|\mathcal{D}_1|}\#_{|\mathcal{D}_1|}^1 \dots s^{m,1}\#_1^m \dots s^{m,|\mathcal{D}_m|}\#_{|\mathcal{D}_m|}^m$ which we denote by t. The $\#_i^k$'s are new symbols that do not occur in any of the databases and serve to mark the end of a string from the respective database.[2] Note that the length of t is $n := \sum_{j=1}^m (\|\mathcal{D}_j\| + |\mathcal{D}_j|)$. The preprocessing then consists of constructing the following data structures for t (in this order): the suffix array SA, the LCP-array LCP, and the information to answer $\text{RMQ}_{\mathsf{LCP}}(i,j)$ in $O(1)$. All steps take time $O(n)$. See Fig. 6.1(a) for an example.

A short definition is necessary at this point: We say that entry $\mathsf{SA}[i]$ *points* to string $s^{j,k}$ in database $\mathcal{D}_j$ iff the first end-of-string marker in $t_{\mathsf{SA}[i]..n}$ is $\#_k^j$. For example, in Fig. 6.1(a), $\mathsf{SA}[8] = 9$ points to $s^{1,2}$, because $t_{9..23} = \texttt{aab}\#_2^1\texttt{bb}\dots$.

[2] The algorithm can actually be implemented such that it uses only m different end-of-string markers, one for each database.

Algorithm 6.1: Labeling of the LCP-array.

Input: suffix array SA and LCP-array LCP of size n for m databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$

Output: m arrays $C'_{\mathcal{D}_1}, \ldots, C'_{\mathcal{D}_m}$ of size n

1 Let $last_{\mathcal{D}_i}$ be an array of size $|\mathcal{D}_i|$, initialized with all 0 ($i = 1, \ldots, m$)
2 Let $C'_{\mathcal{D}_i}$ be an array of size n, initialized with all 0 ($i = 1, \ldots, m$)
3 **for** $i = 1, \ldots, n$ **do**
4 Let j and k be defined such that $\mathsf{SA}[i]$ points to $s^{j,k}$
5 **if** $last_{\mathcal{D}_j}[k] \neq 0$ **then**
6 $l \leftarrow \text{RMQ}_{\mathsf{LCP}}(last_{\mathcal{D}_j}[k] + 1, i)$
7 increase $C'_{\mathcal{D}_j}[l]$ by 1
8 **endif**
9 $last_{\mathcal{D}_j}[k] \leftarrow i$
10 **endfor**

6.3.2 Labeling

Alg. 6.1 augments the LCP-array LCP with arrays $C'_{\mathcal{D}_1}, \ldots, C'_{\mathcal{D}_m}$ which facilitate the computation of the correction terms in phase 3. Although the $C'_{\mathcal{D}_j}$'s are represented by new arrays of size n, we call this step "labeling" because it is derived from the *tree labeling technique* due to Hui (1992). We want $C'_{\mathcal{D}_j}[i]$ to be equal to the number of lexicographically adjacent suffixes (in t) from the *same* string in $\mathcal{D}_j$ that share a longest common prefix of length *exactly* $\mathsf{LCP}[i]$. More formally, $C'_{\mathcal{D}_j}[i]$ equals the number of triples (a, b, k) that fulfill the following constraints:

1. $1 \leq a < i \leq b \leq n$, and $\mathsf{SA}[a]$ and $\mathsf{SA}[b]$ point to the same string $s^{j,k}$.
2. No entry strictly between a and b points to $s^{j,k}$.
3. i is the first position such that $\text{LCE}(a, b) = \mathsf{LCP}[i]$.

The C'-numbers are computed as follows: the for-loop (lines 3–10) scans the suffix array from left to right. Array entry $last_{\mathcal{D}_j}[k]$ holds the rightmost position in SA to the left of i that points to $s^{j,k}$. Thus, if $\mathsf{SA}[i]$ points to $s^{j,k}$, setting $a = last_{\mathcal{D}_j}[k]$ and $b = i$ fulfills constraints 1 and 2 above. As we saw in Sect. 3.3.2, $\text{LCE}(a, b)$ is given by $\text{RMQ}_{\mathsf{LCP}}(last_{\mathcal{D}_j}[k] + 1, i)$ (line 6). See Fig. 6.1(b) for an example.

We now sketch how the C'-numbers help to compute the actual correction terms. We compute $C(\phi)$ for the strings ϕ that are *maximally repeated*. These are strings ϕ that occur x times in t for some value x, but all its proper extensions occur less than x times in t, i.e., ϕa occurs less than x times in t for all $a \in \Sigma$. From the definition of suffix trees, it is obvious that maximally repeated substrings correspond to internal nodes in the suffix tree. And as we saw in Chapter 5, internal nodes in the suffix tree can be identified by an LCP-interval

$[l : r]$, so in order to visit all maximally repeated substrings, our task is to visit all LCP-intervals and calculate the frequency of the respective substrings. It is clear that the frequency of all other strings can be derived from the frequency of the maximally repeated strings.

Now, if $[l : r]$ is the LCP-interval that represents ϕ, with Lemma 6.1 we see that

$$C_{\mathcal{D}_j}(\phi) = \sum_{l<i\leq r} C'_{\mathcal{D}_j}[i] \ . \tag{6.3}$$

The validity of this equation can be seen by comparing the definition of the C'-terms at the beginning of this section with Lemma 6.1: $C'_{\mathcal{D}_j}[i]$ counts how often two lexicographically adjacent strings from the same database $\mathcal{D}_j$ share a longest common prefix of length *exactly* $\mathsf{LCP}[i]$. As the strings in the LCP-interval $[l : r]$ are exactly the strings that are prefixed by ϕ, summing over the C'-values in the interval gives the number of times that ϕ is a prefix of the LCE of two lexicographically adjacent suffixes, the statement of Lemma 6.1.

Eq. (6.3) can be rewritten as

$$C_{\mathcal{D}_j}(\phi) = \sum_{\substack{l<i\leq r \\ \mathsf{LCP}[i]=|\phi|}} C'_{\mathcal{D}_j}[i] \ + \sum_{\substack{[l':r'] \text{ child-interval of } [l:r] \\ [l':r'] \text{ represents } \psi\neq\phi}} C_{\mathcal{D}_j}(\psi) \ ; \tag{6.4}$$

this is due to the fact that an LCP-interval can be split up into the positions of its ℓ-indices and its child-intervals. Thus, (6.4) enables a recursive calculation of the C-terms.

Example 6.3. In Fig. 6.1, the interval $(8, 9)$ (representing $\mathtt{aa}$) gives $C_{\mathcal{D}_1}(\mathtt{aa}) = \sum_{8\leq i\leq 9} C'_{\mathcal{D}_1}[i] = 1 + 0 = 1$, and the interval $(11, 14)$ (representing $\mathtt{ab}$) gives $C_{\mathcal{D}_1}(\mathtt{ab}) = 1 + 0 + 0 + 0 + 0 = 1$. Having this, we can compute $C_{\mathcal{D}_1}(\mathtt{a})$ as $C'_{\mathcal{D}_1}[6] + C'_{\mathcal{D}_1}[7] + C_{\mathcal{D}_1}(\mathtt{aa}) + C'_{\mathcal{D}_1}[10] + C_{\mathcal{D}_1}(\mathtt{ab}) = 1 + 0 + 1 + 2 + 1 = 5$.

Kasai et al. (2001) gave an algorithm that simulates a bottom-up-traversal of the suffix tree by scanning the LCP-array from left to right. We could thus calculate the C-numbers by a modification of their algorithm, applying (6.4) to all LCP-intervals in a bottom-up manner. However, this step can be incorporated into the extraction step (which we explain next), thereby avoiding the need to store the C-numbers in separate arrays.

6.3.3 Extraction

We now describe how to output all strings that pass the frequency-based criterion. As mentioned above, this step is accomplished by a simulated bottom-up-traversal of the suffix tree due to Kasai et al. (2001)[3], calculating for each LCP-interval representing string ϕ the values $S_{\mathcal{D}_j}(\phi)$ and $C_{\mathcal{D}_j}(\phi)$ for $j = 1, \ldots, m$,

[3] Observe that one could also simulate this DFS by means of the Enhanced Suffix Array (cf. Chapter 5). However, as we are doing a bottom-up-traversal here, it is simpler to use Kasai et al.'s method.

Algorithm 6.2: Extraction of all substrings satisfying p.

Input: suffix array SA, LCP-array LCP, $C'_{\mathcal{D}_j}$ as computed by Alg. 6.1 (all of size n), frequency-based predicate $p(\mathsf{supp}_{\mathcal{D}_1}, \ldots, \mathsf{supp}_{\mathcal{D}_m})$

Output: All substrings satisfying p

1 S is a stack holding tuples $(v.h, v.S_{\mathcal{D}_j}, \ldots, v.S_{\mathcal{D}_m}, v.S_{\mathcal{D}_2}, v.C_{\mathcal{D}_1}, \ldots, v.C_{\mathcal{D}_m})$
2 Let v be a stopper element with $v.h = -\infty$, push v on S
3 **for** $i = 1, \ldots, n+1$ **do**
4 $\quad v \leftarrow \mathtt{top}(S)$ $\quad$ {v represents the string to be examined next}
5 $\quad S_{\mathcal{D}_j} \leftarrow 0$, $C_{\mathcal{D}_j} \leftarrow 0$ for all $j = 1, \ldots, m$
6 $\quad$ **while** $v.h > \mathsf{LCP}[i]$ **do**
7 $\quad\quad v \leftarrow \mathtt{pop}(S), w \leftarrow \mathtt{top}(S)$ $\quad$ {w always points to top of stack}
8 $\quad\quad$ **if** $w.h \geq \mathsf{LCP}[i]$ **then**
9 $\quad\quad\quad$ {Otherwise w is not parent node of current node v.}
10 $\quad\quad\quad w.S_{\mathcal{D}_j} \mathrel{+}= v.S_{\mathcal{D}_j}$, $w.C_{\mathcal{D}_j} \mathrel{+}= v.C_{\mathcal{D}_j}$ for all j $\quad$ {accumulate}
11 $\quad\quad$ **endif**
12 $\quad\quad \mathsf{freq}_{\mathcal{D}_j} \leftarrow v.S_{\mathcal{D}_j} - v.C_{\mathcal{D}_j}$ for all $j = 1, \ldots, m$
13 $\quad\quad$ **if** $p(\mathsf{freq}_{\mathcal{D}_1}, \ldots, \mathsf{freq}_{\mathcal{D}_m})$ **then**
14 $\quad\quad\quad$ {Now v represents a maximally repeated substring satisfying p.}
15 $\quad\quad\quad$ **for** $h = \max\{w.h, \mathsf{LCP}[i]\} + 1, \ldots, v.h$ **do** print $t_{\mathsf{SA}[i]..\mathsf{SA}[i]+h-1}$
16 $\quad\quad$ **endif**
17 $\quad\quad S_{\mathcal{D}_j} \leftarrow v.S_{\mathcal{D}_j}$, $C_{\mathcal{D}_j} \leftarrow v.C_{\mathcal{D}_j}$ for all $j = 1, \ldots, m$
18 $\quad\quad v \leftarrow w$
19 $\quad$ **endw**
20 $\quad$ **if** $v.h < \mathsf{LCP}[i]$ **then** push $(\mathsf{LCP}[i], S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, C_{\mathcal{D}_1}, \ldots, C_{\mathcal{D}_m})$ on S
21 $\quad \mathtt{top}(S).C_{\mathcal{D}_j} \mathrel{+}= C'_{\mathcal{D}_j}[i]$ for all $j = 1, \ldots, m$ $\quad$ {gather correction factors}
22 $\quad$ **if** $i \leq n$ **then**
23 $\quad\quad$ Let $\mathsf{SA}[i]$ point to $\mathcal{D}_j$; set $S_{\mathcal{D}_j} \leftarrow 1$ and all other $S_{\mathcal{D}_{j'}}$'s to 0
24 $\quad\quad$ push $(n - \mathsf{SA}[i] + 1, S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, 0, \ldots, 0)$ on S
25 $\quad$ **endif**
26 **endfor**

thereby yielding the frequency of ϕ in $\mathcal{D}_j$ as $S_{\mathcal{D}_j}(\phi) - C_{\mathcal{D}_j}(\phi)$. The formula for the C-numbers is given by (6.4), and for the S-numbers we have

$$S_{\mathcal{D}_j}(\phi) = \sum_{\substack{l \leq i \leq r \\ \mathsf{SA}[i] \text{ points to } \mathcal{D}_j}} 1 \tag{6.5}$$

(again, $[l : r]$ is ϕ's LCP-interval). This is simply because the interval $[l, r]$ in SA represents all suffixes of t that are prefixed by ϕ. As for the C-numbers, (6.5) can be rewritten to allow a recursive calculation:

$$S_{\mathcal{D}_j}(\phi) = \begin{cases} 0 & \text{if } l = r \text{ and } \mathsf{SA}[l] \text{ points to } j' \neq j \\ 1 & \text{if } l = r \text{ and } \mathsf{SA}[l] \text{ points to } j \\ \sum_{\substack{[l':r'] \text{ child-interval of } [l:r] \\ [l':r'] \text{ represents } \psi \neq \phi}} S_{\mathcal{D}_j}(\psi) & \text{otherwise} \end{cases} \tag{6.6}$$

Alg. 6.2 is used for the extraction phase. If one deletes lines 5, 21 and 23 from Alg. 6.2 and substitutes lines 8–17 by the single command "print $t_{\mathsf{SA}[i]..\mathsf{SA}[i]+v.h-1}$," this yields exactly the algorithm in Fig. 7 of Kasai et al. (2001) which solves the *substring traversal problem*, i.e., the enumeration of all maximally repeated substrings. The idea behind this algorithm is to visit all suffixes of t in lexicographic order and to keep all maximally repeated prefixes of the current suffix on a stack S, ordered by their length, with the longest such prefix being on top of S. A more formal description is as follows. Each element on S is represented by a tuple $(h, S_{\mathcal{D}_1}, \ldots, S_{\mathcal{D}_m}, C_{\mathcal{D}_1}, \ldots, C_{\mathcal{D}_m})$, where h is the length of the prefix (i.e., the corresponding prefix is $t_{\mathsf{SA}[i]..v.h-1}$), and the other variables are the counters as defined by (6.1) and (6.2). At the beginning of step i of the for-loop (lines 3–26), we have that the $(i-1)$'st suffix and all maximally repeated prefixes of $t_{\mathsf{SA}[i-1]..n}$ are on S. Then the $(i-1)$'th suffix is visited (line 4) and the following steps are performed:

1. The while-loop (lines 6–19) removes from S all tuples representing strings with length greater than $\text{LCE}(i-1, i) = \mathsf{LCP}[i]$. These are exactly the prefixes of $t_{\mathsf{SA}[i-1]..n}$ which are not a prefix of $t_{\mathsf{SA}[i]..n}$. All strings passing the statistical criterion are returned (line 15).

2. The counter-values $S_{\mathcal{D}_j}(\phi)$ and $C_{\mathcal{D}_j}(\phi)$ of the current string v are added to the respective counters of the string on top of the stack (line 10). This step takes care of the last sums in Eq. (6.4) and (6.6), respectively, as v represents a child of the string which is on top of S.

3. When pushing the longest common prefix of two lexicographically adjacent suffixes on S (line 20), the counter-values are initialized correctly.

4. The C'-numbers are added to the correct string (line 21) which is again on top of the stack. This step takes care of the first sum in (6.4).

5. The suffix $t_{\mathsf{SA}[i]..n}$ is pushed on S with the correct counter-values (lines 22–25). Line 23 accounts for the initialization of the $S_{\mathcal{D}_j}$-values, i.e., the first two cases in (6.6).

It is shown by Kasai et al. (2001) that this algorithm visits all maximally repeated substrings of t, and its running time is $O(n)$ (apart from the for-loop that outputs the solutions, line 11). The discussion from Sect. 6.3.2 shows that the S- and C-values are calculated correctly, and thus in line 9 we have that the frequency of the string ϕ that is represented by v is calculated correctly. We thus have the following

Theorem 6.2 (Frequency-based string mining)**.** *For a constant number of databases of strings of total length n, all strings that satisfy a frequency-based criterion (e.g., emerging substrings) can be calculated in $O(n + s)$ time, solely by using array-based data structures occupying $O(n)$ words of additional space (apart from the output), where s is the total size of the strings that satisfy the criterion.* ■

6.3.4 Reducing the Size of the Output

We have already mentioned that the size of the output can be $\Theta(n^3)$, which may be too big even for realistic problem sizes. In addition, it is often not desirable to have an explicit, humanly readable representation of all strings in the solution space *Th*. Take, for example, the situation where the emerging substrings are to be further processed according to another criterion. In this case it would be much better to have a smaller representation of the emerging substrings, as possibly many of them will not appear in the final output.

It is quite common in such situations to return a *labeled* data structure instead of a complete enumeration of the output, where the labels allow for an easy re-calculation of all patterns in *Th*. For example, Gusfield and Stoye (2004) return a labeled suffix tree which contains enough information to reconstruct all tandem repeats in a string, whose number may already be $\Theta(n^2)$. It is not difficult to come up with a similar approach for our problem setting. As the frequencies of each string ϕ is the same as the frequency of the shortest maximally repeated substring that is prefixed by ϕ, all we have to do is to attach a label to each maximally repeated substring that passes the frequency-based predicate, instead of executing the for-loop in line 15 of Alg. 6.2. A space-efficient choice for this label is to mark the first ℓ-index of a pattern in *Th* with a single bit (for example, the sign bit in LCP). It has been shown by Abouelhoda et al. (2004) that the ℓ-intervals can be computed as a by-product of the simulated suffix-tree traversal, and we have seen in Sect. 5.3 that the first ℓ-index can be found by a single range minimum query. Thus, we get the following variant of Thm. 6.2:

Theorem 6.3. *For a constant number of databases of strings of total length n, a* representation *of all strings that satisfy a frequency-based criterion can be calculated in $O(n)$ time, solely by using array-based data structures occupying $O(n)$ words of additional space.* ■

Table 6.1: Datasets used for evaluating the practical performance of the different methods for mining frequent substrings.

subset	# species	size in kB	comment
all	25,734	43,515	complete ARB
prok	14,000	23,011	subset of all
bact	13,657	22,281	subset of prok
proteo	6062	9,867	subset of bact
β-γ	3878	6,343	subset of proteo
β	1151	1,860	subset of β-γ
β_{59}	59	88	subset of β
xanthogr	143	235	subset of proteo
xanthom	59	91	subset of xanthogr
random	60	100	subset of all

6.4 Practical Performance

The aim of this section is to show that our new method also works fast in practice, even on large datasets. We implemented the algorithm from Sect. 6.3 in C++ (available at `www.bio.ifi.lmu.de/~fischer/`) so that it finds frequent substrings (Problem 1) and emerging substrings (Problem 2), respectively. For the construction of the suffix array we used the method due to Manzini and Ferragina (2004), and for the RMQ-preprocessing we used the "engineered"-implementation from Sect. 3.8, because space is a primary issue in large-scale data mining.

Unfortunately, because an implementation of the emerging substring miner (Chan et al., 2003) is not publicly available, we could only run comparative tests for mining frequent substrings. We compared our method to the algorithms called VST (De Raedt et al., 2002) and FAVST (Lee and De Raedt, 2005).[4] The "VST" in both names is an abbreviation of *Version Space Tree*, which are simply suffix *tries* with some additional satellite information. All tests were performed on an Athlon XP 3300 with 2GB of RAM under Linux. All programs were compiled with `g++`, using the options "-O3 -fomit-frame-pointer -funroll-loops." Further, instead of writing the output to disk, all programs were adapted to redirect their output to a virtual "null"-device called `/dev/null` under Linux, in order to eliminate the influence of the access time to secondary storage units.

We used the Jan'03 release of the nucleotide database from the ARB project (Ludwig et al., 2004), containing rRNA of about 25,000 species. Because rRNA is highly preserved in evolution, the data bears a high sequential similarity, and the running time of all programs should thus be highly influenced by the chosen input parameters. A phylogenetic tree partitions the species into different

[4] We wish to thank Sau Dan Lee for providing the source codes of his methods FAVST and VST.

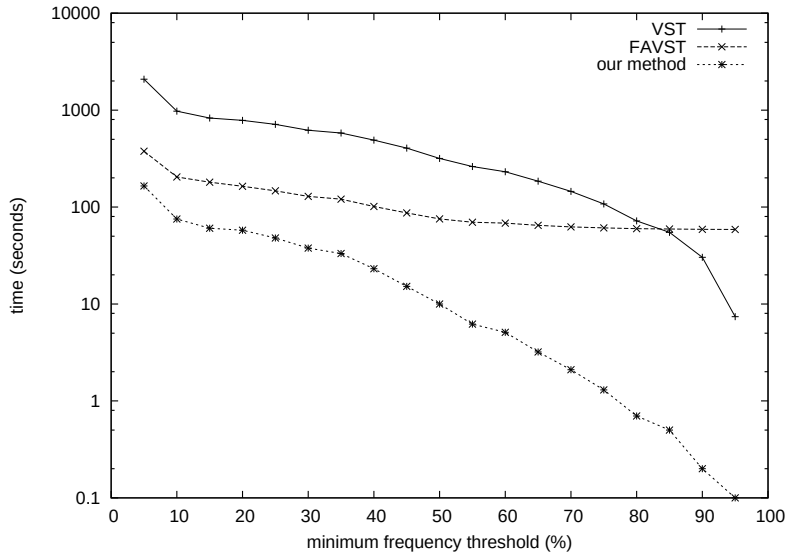


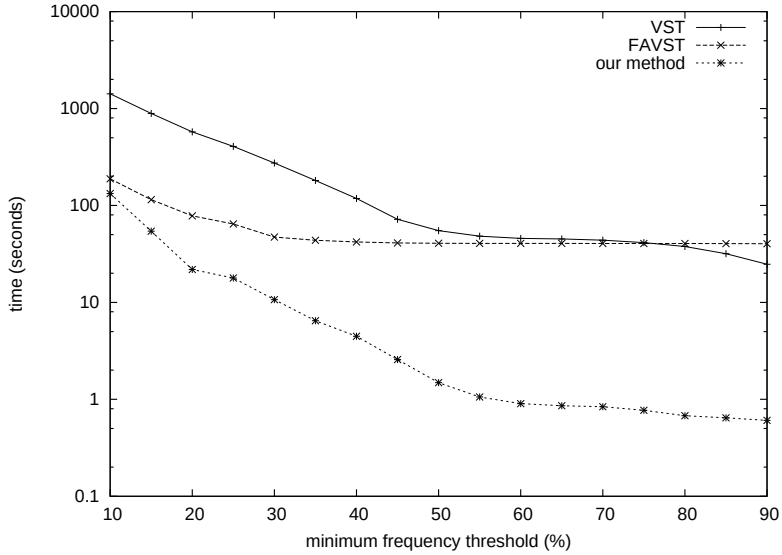
Figure 6.2: Comparison of the three methods for a single minimum frequency query. Note the logarithmic y-scale.

groups, of which we selected some for evaluation. The subsets used are shown in Tbl. 6.1.

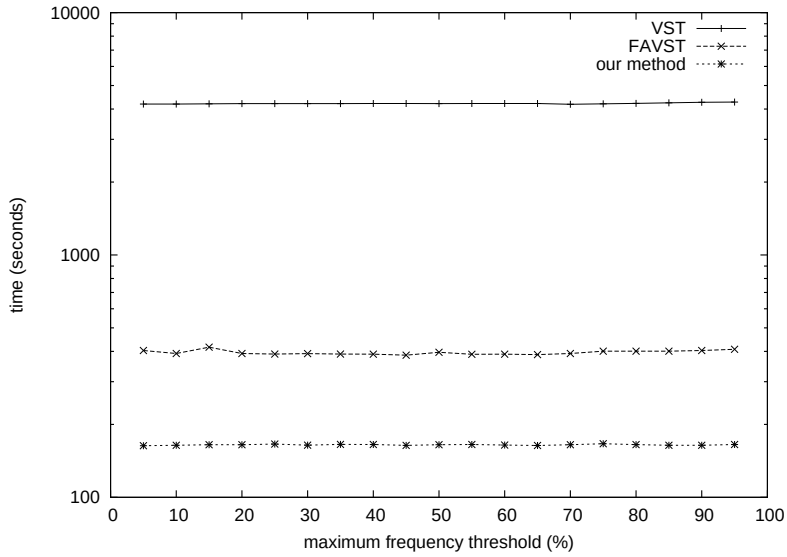
For the comparison with FAVST and VST, we were forced to pick a very small subset of the ARB-database. This is because algorithm FAVST builds a non-compacted suffix trie on the *whole* database, which has size $O(n^2)$ in the worst case. As this is *extremely* space consuming, we could only use FAVST for subsets of size less than 100kB. Already for datasets of this size the space consumption of FAVST is more than 1.5GB. The reason why VST cannot be applied to larger datasets is that it is incredibly slow — we will see presently that for instances where our method takes only about 2 minutes, VST already needs more than one hour!

The first test was a single minimum frequency query on 60 random entries of the database. The results, for varying values of *min*, can be seen in Fig. 6.2. It is striking that our method is faster than both FAVST and VST, sometimes by several orders of magnitude. Further, it is interesting to see that the running time of FAVST does not drop as much as the other methods with increasing values of *min*, as it is the case for VST and our method. This is because constructing the suffix trie for the whole database is the most time consuming part of the algorithm and is independent of *min*. Further tests with other random subsets of the complete ARB-database revealed similar results.

In a second test we wanted to test the performance of a combined minimum- and maximum frequency query. For this, we selected two disjoint subsets with a higher biological relevance. The dataset chosen for the minimum frequency criterion was xanthom, and for maximum frequency criterion we chose a subset of the β-dataset, called β_{59} (again, the space consumption of FAVST forced



(a) This graph shows the dependency on the minimum frequency threshold.



(b) This graph shows the dependency on the maximum frequency threshold.

Figure 6.3: Comparison of the three methods for a combined minimum- and maximum-frequency query. Note the logarithmic y-scales.

size (MB)	number of proteins	time (min)
25	96,939	1:10
50	186,913	2:58
75	269,590	5:03
100	359,504	7:17
125	458,088	9:20
150	529,585	11:32

Table 6.2: Results for a minimum frequency query on files containing amino acid sequences from the Swissprot database.

us to pick such small datasets). The results for varying values of *min* can be seen in Fig. 6.3(a), where the maximum frequency threshold was held fixed at 50%. They resemble those from the previous experiments, except that for small values of *min* our method and FAVST perform about equally well (but still with a factor-2-advantage for our method). Profiling showed that the sheer amount of frequent patterns to be written on disk was the most time consuming part in these cases, which cannot be avoided by any method.

Figure 6.3(b) shows the results for the same test, but now for different values of *max*. The value for the minimum frequency query was held fixed at 3%, in order not to filter out too many patterns already by the minimum frequency constraint. However, even with this very lax choice of *min* the running times do not depend as much on *max* for all three methods as in the case of Fig. 6.3(a). Still, our method is always the fastest, with a factor of about 4 compared to FAVST, and a factor of more than 25 compared to VST.

In a last test we wanted to evaluate the scalability of our method. We created several files containing the first 25, 50, ..., 150MB of the file "proteins" from the Pizza & Chili-site (Ferragina and Navarro, 2005), which contains amino acid sequences of various species obtained from the Swissprot database. Our implementation of the pure minimum frequency miner uses 3 integer arrays of size n, one for the suffix- and LCP-array, and one for the C'-counters. This is the reason why 150MB was the largest file we could process on a 2GB ($150\text{MB}\cdot 4\cdot 3 = 1,800\text{MB}$).

We then performed a simple minimum frequency query, with a minimum frequency threshold of 1% of the number of sequences in the database. The results of this test can be seen in Tbl. 6.2. It can be seen that despite the very low minimum frequency threshold of 1% the running times are quite fast. This is because proteins do not exhibit such a high sequential similarity as rRNA does, so there are less patterns in the solution space. Hence, the running time of our algorithm is not that much dominated by the time needed to write the output (as in the previous experiments).

6.5 Conclusions

We presented a theoretically optimal solution to string mining under frequency constraints. We mostly built upon results on index structures for strings. One of the building blocks is the fast computation of range minimum queries. Given this algorithmic framework, it is possible to compute solutions, e.g., for emerging substrings and patterns statistically associated with classes of sequences, very efficiently. We have seen in the experiments that our method surpasses previous approaches by all means, and also that it is scalable to biological databases of realistic size.

CHAPTER

7

Suffix Arrays on Words

7.1 Chapter Introduction

As we have already seen in Chapter 2, one of the most important tasks in classical string matching is to construct an *index* on the input text in order to answer future queries faster. Well-known examples of such indexes include suffix-trees, word graphs, and suffix arrays (see, e.g., Gusfield, 1997). Despite the extensive research that has been done in the last three or four decades, this topic has recently re-gained popularity with the rise of compressed indexes (see Navarro and Mäkinen, 2007) and new applications such as data compression, text mining, and computational linguistics.

However, all of the indexes mentioned above are *full-text* indexes, in the sense that they index any position in the text and thus allow to search for occurrences of patterns starting at *arbitrary* text positions. In many situations, deploying the full-text feature might be like using a "cannon to shoot a fly," with undesired negative impacts on both query time and space usage. For example, in European languages, words are separated by special symbols such as spaces or punctuation signs; in a dictionary of URLs, "words" are separated by dots and slashes. In both cases, the results found by a word-based search with a full-text index would have to be filtered out by discarding those that do not occur at word-boundaries. Possibly a time-costly step! Additionally, indexing every text position would affect the overall space occupancy of the index, with an increase in the space complexity which could be estimated in practice as a factor 5–6, given the average word length of linguistic texts. Of course, the use of word-based indexes is not limited to pattern searches, as they have been successfully used in many other contexts, like data compression (Isal and Moffat, 2001) and computational linguistics (Yamamoto and Church,

2001), just to cite a few.

Surprisingly enough, *word-based* indexes have been introduced only recently in the string-matching literature (Andersson et al., 1999), although they were very famous in Information Retrieval many years before (cf. Witten et al., 1999). The basic idea underlying their design consists of storing just a *subset* of the text positions, namely the ones that correspond to word beginnings. It is actually easy to construct such indexes if $O(n)$ additional space is allowed at construction time (n being the text size): Simply build the normal index for every position in the text and then discard those positions which do not correspond to word beginnings. Unfortunately, such a simple (and common, among practitioners!) approach is not space optimal. In fact, $O(n)$ construction time cannot be improved, because this is the time needed to scan the input text. But $O(n)$ additional working space (other than the indexed text) seems too much because the final index will need $O(k)$ space, where k is the number of words in the indexed text. This is an interesting issue, not only theoretically, because "*... we have seen many papers in which the index simply 'is,' without discussion of how it was created. But for an indexing scheme to be useful it must be possible for the index to be constructed in a reasonable amount of time*" (Zobel et al., 1996). And in fact, the working-space occupancy of construction algorithms for full-text indexes is yet a primary concern and an active field of research (Hon et al., 2003).

The first result addressing this issue in the word-based indexing realm is due to Andersson et al. (1999), who showed that the so-called *word suffix tree* can be constructed in $O(n)$ *expected* time and deterministic $O(k)$ working space. In 2006, Inenaga and Takeda (2006a) improved this result by providing an on-line algorithm which runs in $O(n)$ time in the worst case and $O(k)$ space in addition to the indexed text. They also gave two alternative indexing structures (Inenaga and Takeda, 2006b,c) which are generalizations of Directed Acyclic Word Graphs (DAWGs) or compact DAWGs, respectively. The compact version has the same worst case guarantees as the suffix tree, though being smaller in practice. All of Inenaga and Takeda's construction methods are variations of the construction algorithms for (usual) suffix trees (Ukkonen, 1995), DAWGs (Blumer et al., 1985) and CDAWGs (Inenaga et al., 2005), respectively.

The only missing item in this quartet is a word-based analog of the suffix array, a gap which we close in this chapter. We emphasize the fact that, as it is the case with full-text suffix arrays (see, e.g., Kärkkäinen et al., 2006), we get a class-note solution which is simple and practically effective, thus surpassing the previous ones by all means.

A comment is in order at this place. A more general problem than word-based string matching is that of *sparse* string matching, where the set of points to be indexed is given as an arbitrary set, not necessarily coinciding with the word boundaries. Although Inenaga and Takeda (2006a,b,c) claim that their indexes can solve this task as well, they did not take into account that search time becomes exponential in the pattern length in this case.[1] To the best of

[1] This fact has been acknowledged by Shunsuke Inenaga (personal communication, Dec. 2006).

our knowledge, this problem is still open. The only step in this direction has been made by Kärkkäinen and Ukkonen (1996) who considered the special case where the indexed positions are evenly spaced.

7.2 Chapter Outline

We define a new data structure called the *word(-based) suffix array* and show how it can be constructed *directly* in optimal time and space; i.e., without first constructing the sparse suffix tree. The size of the structure is k RAM words, and at no point during its construction more than $O(k)$ space (in addition to the text) is needed. This is also interesting in theory because we could compress the text with Ferragina and Venturini's method (2007) and then build the word-based index in space $O(k \log n) + nH_h + o(n)$ bits (including the text) and $O(n)$ time, simultaneously over all $h = o(\log n)$, where H_h is the h'th order empirical entropy of the indexed text (alphabet is assumed to have constant size). If the number k of indexed "words" is relatively "small," namely $k = o(n/\log n)$, this index would take the same space as the best compressed indexes (see again Navarro and Mäkinen, 2007), namely $nH_h + o(n)$ bits, but it would need less space to be constructed.

As far as pattern-queries are concerned, it is easy to adapt to our word-based suffix array the classical pattern searches over full-text suffix arrays. For patterns of length m, we can easily show that *counting* queries take $O(m \log k)$ time, or $O(m + \log k)$ if an additional array of size k is used. Note that this reduces the number of costly binary search step by $O(\log(n/k))$ compared with full-text suffix arrays. We further show that it is possible to adapt the ideas from Chapter 5, thereby lowering the time bounds for counting queries to $O(m|\Sigma|)$ and $O(m \log |\Sigma|)$, respectively. *Reporting* queries take $O(occ)$ additional time, independent of the search method being used, where occ is the number of word occurrences reported.

In order to highlight the simplicity, and hence practicability, of our approach, we extensively test our different searching methods over various data sets, which cover some typical applications of word-based indexes: natural and artificial language, structured data and prefix-search on hosts/domains in URLs. The experimental results are reported in Section 7.7, there we show that word-based suffix arrays are better than (filtered) full-text suffix arrays in terms of both time and space of the construction and the search phases: Construction time is twice as fast as state-of-the-art algorithms applied on full-text suffix arrays, and the working space is lower than 20%; query time is faster by up to a factor of three *without* post-filtering the word-aligned occurrences. As can be expected, *including* the post-filtering makes it slower by 1–5 orders of magnitude (!), depending on the number of pattern occurrences; so this idea is excluded from further discussion already at this point.

Our result smoothes the way for the deep investigation of word-based Burrows-Wheeler compressors (Isal and Moffat, 2001) and indexes (the latter being not much investigated yet!), and for the engineering of computational linguistics and

text mining tools based on n-grams statistics of very large document collections (Yamamoto and Church, 2001).

7.3 Definitions

Throughout this chapter let T be a text of length n over a an alphabet Σ. We further assume that certain characters from a constant-sized subset W of the alphabet act as *word boundaries*, thus dividing T in a natural sense into k tokens, hereafter called *words*. In Western languages one could think of spaces and punctuation as being part of W; in URL dictionaries one set $W = \{., /\}$. Now let I be the set of positions where new words start: $1 \in I$ and $i \in I \setminus \{1\} \iff T_{i-1} \in W$. (The first position of the text is always taken to be the beginning of a new word.) Similar to Inenaga and Takeda (2006a) we define the set of all suffixes of T starting at word boundaries as $\mathit{Suffix}_I(T) = \{T_{i..n} : i \in I\}$. Then the *word suffix array* $A[1..k]$ is a permutation of I such that $T_{A[i-1]..n} < T_{A[i]..n}$ for all $1 < i \leq k$; i.e., A represents the lexicographic order of all suffixes in $\mathit{Suffix}_I(T)$. We are now ready to state what we mean by *searching* in a word suffix array (note the analogy to Def. 2.1):

Problem 7.1 (Word Aligned String Matching). For a given pattern P of length m let $O_P \subseteq I$ be the set of word-aligned positions where P occurs in T: $i \in O_P$ iff $T_{i..n}$ is prefixed by P and $i \in I$. Then the tasks of word aligned string matching are (1) to answer whether or not O_P is empty *(decision query)*, (2) to return the size of O_P *(counting query)*, and (3) to enumerate the members of O_P in some order *(enumeration query)*.

7.4 Optimal Construction of the Word Suffix Array

This section describes the optimal $O(n)$ time and $O(k)$ space algorithm to construct the word suffix array. For simplicity, we describe the algorithm with only one word separator (namely #). The reader should note that all steps are valid and can be computed in the same time bounds if we have more than one (but constantly many) word separators. We also assume that the set I of positions to be indexed is implemented as an increasingly sorted array. I will certainly be in this form if built when scanning the input string once from left to right.

The algorithm can be summarized in the following four steps:

1. Sort the suffixes from $\mathit{Suffix}_I(T)$ up to the next # (i.e., based on their first word).
2. Build a new text T' from the bucket-numbers obtained in step 1.
3. Sort all suffixes from T' with a linear-time algorithm (Kim et al., 2005; Ko and Aluru, 2005; Kärkkäinen et al., 2006).

bucket–>	**1**			**2**		**3**	**4**		**5**	
	1	2	3	4	5	6	7	8	9	10
A=	4	9	22	6	24	18	1	11	14	27
	a	a	a	a	a	a	a	a	b	b
	#	#	#	a	a	a	b	b	a	a
	a	a	a	#	#	b	#	#	a	a
	a	b	a	a	b	#	a	b	#	#
	#	#	#	#	a	a	#	a	a	
	a	b	b	a	a	#	a	a	a	
	#	a	a	b	#	a	a	#	b	
	.	.	.	.		.	.	.	.	
	.	.	.	.		.	.	.	.	
	.	.	.	.		.	.	.	.	

Figure 7.1: The initial radix-sorting from step 1.

4. From the suffix array of T', derive the word suffix array of T.

We now describe these four steps in more detail. As a running example we use $T = ab\#a\#aa\#a\#ab\#baa\#aab\#a\#aa\#baa\#$, so $I = [1, 4, 6, 9, 11, 14, 18, 22, 24, 27]$.

1. The task of this step is to establish a "coarse" sorting of the suffixes from $\mathit{Suffix}_I(T)$. In particular, we want to sort these suffixes using their *first word* as the sort key. To do so, initialize the array $A[1..k] = I$. Then radix-sort the elements in A: at each level $l \geq 0$, bucket-sort the array A using $T_{A[i]+l}$ as the sort key for $A[i]$. Stop the recursion when a bucket contains only one element, or when a bucket consists only of suffixes starting with $w\#$ for some $w \in (\Sigma \setminus \{\#\})^\star$. Since each character from T is involved in at most one comparison, this step takes $O(n)$ time. See Fig. 7.1 for an example, where (for illustrative purposes) we show below each position i an initial portion of the corresponding suffix $T_{A[i]..n}$, which is of course not explicitly represented in the actual algorithm.

2. We wish to refine the initial sorting of step 1 by using known linear-time algorithms for (full-text) suffix arrays. To prepare for this step, we construct a new text $T' = b(I[1])b(I[2])\ldots b(I[k])$, where $b(I[i])$ is the bucket-number (after step 1) of suffix $T_{I[i]..n} \in \mathit{Suffix}_I(T)$. In our example, $T' = \mathbf{4121453125}$. (We use boldface letters to emphasize the fact that we are using a new alphabet here.) This step can clearly be implemented in $O(k)$ time.

3. We now build the (full-text) suffix array SA for T'. Because all linear-time construction algorithms for suffix arrays (Kim et al., 2005; Ko and Aluru, 2005; Kärkkäinen et al., 2006) work for integer alphabets, we can employ any of them to take $O(k)$ time. See Fig. 7.2 for an example. In this figure, we have attached to each position in the new text T' the corresponding position in T as a superscript (i.e., the array I), which will be useful in the next step.

4. This step derives the word suffix array A from SA. Scan SA from left to

	1^{1}	2^{4}	3^{6}	4^{9}	5^{11}	6^{14}	7^{18}	8^{22}	9^{24}	10^{27}
T'=	**4**	**1**	**2**	**1**	**4**	**5**	**3**	**1**	**2**	**5**
SA=	2	8	4	3	9	7	1	5	10	6
	1	**1**	**1**	**2**	**2**	**3**	**4**	**4**	**5**	**5**
	2	**2**	**4**	**1**	**5**	**1**	**1**	**5**		**3**
	1	**5**	**5**	**4**		**2**	**2**	**3**		**1**
	4		**3**	**5**		**5**	**1**	**1**		**2**
	5		**1**	**3**			**4**	**2**		**5**
	3		**2**	**1**			**5**	**5**		
	1		**5**	**2**			**3**			
	2			**5**			**1**			
	5						**2**			
							5			

Figure 7.2: The new text T' and its (full-text) suffix array SA.

	1	2	3	4	5	6	7	8	9	10
A=	4	22	9	6	24	18	1	11	27	14
	a	a	a	a	a	a	a	a	b	b
	#	#	#	a	a	a	b	b	a	a
	a	a	a	#	#	b	#	#	a	a
	a	a	b	a	b	#	a	b	#	#
	#	#	#	#	a	a	#	a		a
	a	b	b	a	a	#	a	a		a
	#	a	a	b	#	a	a	#		b
	.	.	.	.	.	.	.	.		.
	.	.	.	.	.	.	.	.		.
	.	.	.	.	.	.	.	.		.

Figure 7.3: The final word suffix array for our example text.

right and write the corresponding suffix to A: $A[i] = I[\mathsf{SA}[i]]$. This step clearly takes $O(k)$ time. See Figure 7.3 for an example.

We summarize the overall result in the following theorem.

Theorem 7.1 (Optimal construction of the word suffix array)**.** *Given a text T of length n consisting of k words, the word suffix array for T can be constructed directly in optimal $O(n)$ time and $O(k)$ extra space, using only 3 integer arrays of length k apart from the text.*

Proof. Time and space bounds have already been discussed in the description of the algorithm; it only remains to prove the correctness. This means that we have to prove $T_{A[i-1]..n} \leq T_{A[i]..n}$ for all $1 < i \leq k$ after step 4. Note that after step 1 we have $T_{A[i-1]..x} \leq T_{A[i]..y}$, where x and y are defined s.th. T_x and T_y is the first # after $T_{A[i-1]}$ and $T_{A[i]}$, respectively. We now show that steps 2–4 refine this ordering for buckets of size greater than one. In other words, we wish to show that in step 4, buckets $[l : r]$ sharing a common prefix $T_{A[i]..x}$, with T_x being the first # for all $l \leq i \leq r$, are sorted using the lexicographic order of $T_{x+1..n}$ as a sort key. But this is simple: because the newly constructed text T' from step 2 respects the order of $T_{A[i]..x}$, and because step 3 establishes the correct lexicographic order of T', the $I[\mathsf{SA}[i]]$'s are the correct sort keys for step 4. ■

We mention that establishing T' (steps 1 and 2 above) could also be accomplished by building a trie over the input words, then assigning integer ranks to the words by traversing it in lexicographic order, and finally substituting the words by these ranks (cf. Andersson et al., 1999); however, we opted for the idea based on radix-sort because the algorithm can then be implemented using only 3 integer arrays of size k in addition to T (which takes n bytes). Additionally, this approach would counteract our aim to avoid dynamic data structures.

To further reduce the required space, we can think of compressing T before applying the above construction algorithm by adopting an entropy-bounded storage scheme for strings (Sadakane and Grossi, 2006; González and Navarro, 2006; Ferragina and Venturini, 2007) which allows constant-time access to any of its $O(\log n)$ contiguous bits. This implies the following:

Corollary 7.2. *The word suffix array can be built in $3k \log n + nH_h(T) + o(n)$ bits and $O(n)$ time. For any $k = o(n/\log n)$, the space needed to store this data structure is $nH_h + o(n)$ bits, simultaneously over all $h = o(\log n)$.* ■

This result is interesting because it says that in the case of a tokenized text with long words on average (e.g., a dictionary of URLs), the word-based suffix array takes the same space as the best compressed (full-text) indexes (see Navarro and Mäkinen, 2007), but it would need less space to be constructed.

7.5 Word-Based LCP-Arrays

This section defines the word-based LCP-array and shows that it can be computed in $O(n)$ time and $O(k)$ extra space. Apart from improving the pattern matching algorithms in the following section, enhancing the word suffix array with the word-based LCP-array yields all the functionalities of the (full-text) enhanced suffix array. E.g., with the LCP-array it is possible to simulate bottom-up traversals of the corresponding word suffix tree (Kasai et al., 2001), and augmenting this further with RMQ-information allows us also to simulate top-down traversals (Thm. 5.4). Additionally, in the vein of Aluru (2006, Section 5.3.2), we can derive the word suffix *tree* from LCP and A. This yields a simple, space-efficient and memory-friendly (in the sense that nodes tend to be stored in the vicinity of their predecessor/children) alternative to the algorithm from Inenaga and Takeda (2006a), stated in the following

Corollary 7.3 (Constructing word suffix trees via word suffix arrays)**.** *Given a text T of length n consisting of k words, the word suffix tree for T can be constructed in optimal $O(n)$ time and $O(k)$ extra space if one first constructs the word suffix array and the word-based LCP-array.* ■

Let us first formally define the *LCP-array* $\mathsf{LCP}[1, k]$ as follows: $\mathsf{LCP}[1] = 0$ and for $i > 1$, $\mathsf{LCP}[i]$ is the length of the longest common prefix of the suffixes $T_{A[i-1]..n}$ and $T_{A[i]..n}$. For our example string, the LCP-array is $\mathsf{LCP} = [0, 5, 3, 1, 3, 2, 1, 3, 0, 4]$. We will now show that this LCP-table can be computed

Algorithm 7.1: Word-based $O(n)$-time longest common prefix computation using $O(k)$ space (adapted from Kasai et al., 2001).

```
1  LCP[1] ← −1, h ← 0
2  for i ← 1, ..., k do
3      p ← A^{-1}[i], h ← max{0, h − A[p]}
4      if p > 1 then
5          while T_{A[p]+h} = T_{A[p−1]+h} do
6              h ← h + 1
7          endw
8          LCP[p] ← h
9      endif
10     h ← h + A[p]
11 endfor
```

in $O(n)$ time in the order of *inverse word suffix array* A^{-1} which is defined as $A[A^{-1}[i]] = I[i]$; i.e., $A^{-1}[i]$ tells us where the i'th-longest suffix among all *indexed* suffixes from T can be found in A. A^{-1} can be computed in $O(k)$ time as a by-product of the construction algorithm (Section 7.4). In our example, $A^{-1} = [7, 1, 4, 3, 8, 10, 6, 2, 5, 9]$.

Alg. 7.1 shows how to compute the LCP-array in $O(n)$ time. It is actually a generalization of the $O(n)$-algorithm for LCP-computation in (full-text) suffix arrays (Kasai et al., 2001). The difference from that algorithm is that the original algorithm assumes that when going from position p (here $A[p] = I[i]$) to position $p' = A^{-1}[i+1]$ (hence $A[p'] = I[i+1]$), the difference in length between $T_{A[p]..n}$ and $T_{A[p']..n}$ is exactly one (cf. Prop. 2.4), whereas in our case this difference may be larger, namely $A[p'] - A[p]$. This means that when going from position p to p' the lcp can decrease by at most $A[p'] - A[p]$ (instead of 1); we account for this fact by adding $A[p]$ to h (line 10) and subtracting p' (i.e., the new p) in the next iteration of the loop (line 3). At any iteration, when entering line 4, variable h holds the length of the prefix that $T_{A[p]..n}$ and $T_{A[p-1]..n}$ are guaranteed to have in common. Since each text character is involved in at most 2 comparisons, the $O(n)$ time bound easily follows.

Theorem 7.4 (Optimal construction of the word-based LCP-array)**.** *The word-based LCP-array* $\mathsf{LCP}[1, k]$ *for a text* T *of length* n *consisting of* k *words can be constructed in-place in optimal* $O(n)$ *time.*

Proof. Follows directly from the discussion above and the fact that the trick for in-place LCP-array-construction (i.e., linking the suffixes from $\mathit{Suffix}_I(T)$ in order of their appearance in A and overwriting these links while constructing LCP) can also be applied to our algorithm (Manzini, 2004). ■

As $\mathsf{LCP}[A^{-1}[1]] + I[1], \mathsf{LCP}[A^{-1}[2]] + I[2], \ldots, \mathsf{LCP}[A^{-1}[k]] + I[k]$ is again an increasing sequence of k integers in the range $[1, n]$, we can also apply the compression trick from Prop. 2.5, yielding the following

method	space usage (words)	time bounds
bin-naive	k	$O(m \log k + occ)$
bin-improved	$(1+c)k,\ c \leq 1$	$O((m - \log(ck)) \log k + occ)$
bin-lcp	$2k$	$O(m + \log k + occ)$
esa-search	$2k + O(k/\log k)$	$O(m\lvert\Sigma\rvert + occ)$
esa-log	$2k + O(k/\log k)$	$O(m \log \lvert\Sigma\rvert + occ)$

Table 7.1: Different methods for retrieving all *occ* occurrences of a pattern at word-boundaries. The full-text suffix array would have the same time- and space-bounds, with k substituted by $n \gg k$, and *occ* by $occ' \gg occ$, where occ' is the number of not necessarily word-aligned occurrences of the pattern (i.e., before the post-filtering stage).

Corollary 7.5 (Succinct encoding of the word-based LCP-array)**.** *The word-based LCP-array for a text of length n consisting of k words can be stored in $n + k + o(n)$ bits.*

Proof. Similar to Prop. 2.5, but now the bit vector H consists of k ones and $\leq n$ zeros. Preparing H for constant-time rank and select needs another $o(n)$ bits. ■

7.6 Searching in the Word Suffix Array

We now consider how to search for the *occ* word-aligned occurrences of a pattern $P[1, m]$ in the text $T[1, n]$. Due to the lexicographic order of the word suffix array A, suffixes from $\mathit{Suffix}_I(T)$ sharing a common prefix form an interval in A. This means that in order to solve tasks 1–3 from Probl. 7.1 it suffices to find an interval $[l : r]$ in A such that, for $l \leq i \leq r$, $T_{A[i]..n}$ are exactly those suffixes from $\mathit{Suffix}_I(T)$ that are prefixed by P: For the decision and counting tasks we check the size of the interval, and for the the enumeration task we return the set $\{A[l], \ldots, A[r]\}$ of all *occ* occurrences in additional $O(occ)$ time. As searching the word suffix array can be done with the same methods as in the full-text suffix array we keep the discussion short (see also Table 7.1); the purpose of this section is the completeness of exposition, and to prepare for the experiments in the following section.

7.6.1 Searching in $O(m \log k)$ Time

Because A is sorted lexicographically, it can be binary-searched in a similar manner to the original search-algorithm from Manber and Myers (1993). Because this algorithm is the basis for the remainder of this section, we sketch it in Alg. 7.2. This algorithm needs actually be called twice: once for searching the left border l as it is shown in the figure, and a second time when search-

Algorithm 7.2: How to locate the leftmost index of P in A in $O(m \log k)$ time. As a by-product, the algorithm also calculates the initial search interval $[l_1 : r_1]$ for the rightmost index (lines 7–9).

Input: pattern P of length m
Output: P's leftmost index l in A

1 **if** $P \leq T_{A[1]..m}$ **then return** 1
2 **else if** $P > T_{A[k]..m}$ **then return** $k+1$
3 **else**
4 $\quad (l, r) \leftarrow (1, k), found \leftarrow$ `false`
5 $\quad$ **while** $r - l > 1$ **do**
6 $\quad\quad M \leftarrow (l + r)/2$
7 $\quad\quad$ **if** $P = T_{A[M]..m} \wedge found =$ `false` **then**
8 $\quad\quad\quad l_1 \leftarrow M, r_1 \leftarrow r, found \leftarrow$ `true` {start-interval for next call}
9 $\quad\quad$ **endif**
10 $\quad\quad$ **if** $P \leq T_{A[M]..m}$ **then** $r \leftarrow M$
11 $\quad\quad$ **else** $l \leftarrow M$
12 $\quad$ **endw**
13 **endif**
14 **return** r {this is the left boundary!}

ing for the right border r. In the latter case, the search interval is initialized with $[l_1, r_1]$ instead of $[1, k]$ (line 4); these values have been computed as a by-product during the search for l (lines 7–9). The resulting search algorithm is called *bin-naive* from now on.

We can also apply the two heuristics proposed by Manber and Myers (1993) to speed up the search in practice, though not in theory (the resulting search algorithm is called *bin-improved*). The first of these heuristics is used to narrow down the initial search interval in A. It builds an additional array B of size $|\Sigma|^\ell$ (ℓ to be defined presently), where $B[a]$ points to the first entry i in A s.th. $T_{A[i]..n}$ is prefixed by $a \in \Sigma^\ell$. Then the search for P can certainly be limited to the interval $[B[P_{1..\ell}] : B[P_{1..\ell} + 1]]$ in A, where $P_{1..\ell} + 1$ denotes the lexicographically next string of $P_{1..\ell}$. In practice it is advisable to choose $\ell = \log_{|\Sigma|}(ck)$ for a constant $c \leq 1$; then the additional space needed is bounded by k.

The second heuristic reduces the number of character comparisons by remembering the number of matching characters from T and P that have been seen so far. Imagine the current search interval in A is $[l : r]$, so we want to compare P to $T_{A[(l+r)/2]..n}$. Instead of starting the comparison at each step from scratch, suppose we know that l' initial characters of P match with $T_{A[l]..n}$, and r' characters with $T_{A[r]..n}$. Because of the lexicographic ordering of A we then know that the first $\min\{l', r'\}$ characters of P match with $T_{A[(l+r)/2]..n}$, so these comparisons can be saved. As the values l' and r' are obtained as a by-product of the string-matching (lines 7/10) during the binary search, this second heuristic constitutes no extra work.

7.6.2 Searching in $O(m + \log k)$ Time

Like in the original article by Manber and Myers (1993) the idea is to precompute the longest common prefixes of $T_{A[(l+r)/2]..n}$ with both $T_{A[l]..n}$ and $T_{A[r]..n}$ for all possible search intervals $[l : r]$. The key insight for this technique is that there are only k possible such search intervals (imagine a perfect binary tree on top of A!), so these precomputations only use linear time and space. Footnote 6 in their article actually shows that only one of these values needs to be stored, so the additional space needed is one array of size k. We refer the reader to the original article for a complete description of the algorithm (which we call *bin-lcp* from now on).

With the word-based LCP-array we also have a succinct version of the $O(m+\log k)$-time pattern search: compute LCP and $\text{RMQ}_{\mathsf{LCP}}$, using a total of $n+3k+o(n)$ bits (Cor. 7.5 and Thm. 3.11). These two tables give access to the longest common prefixes needed in the binary search (because they give access to *all* longest common prefixes).

7.6.3 Searching in $O(m|\Sigma|)$ and $O(m \log |\Sigma|)$ Time

While the previous two searching algorithms have a searching time that is *independent* of the alphabet size, we show in this section how to locate the interval of P in A in $O(m|\Sigma|)$ time. We note that in the special but (at least theoretically) highly relevant case where the alphabet size is constant this actually yields *optimal* $O(m)$ counting time and *optimal* $O(m + occ)$ enumeration time.

The techniques for searching are similar to the ones presented in Chapter 5: in order to achieve $O(m|\Sigma|)$ matching time, we use the LCP-array from the previous section, plus the RMQ-information on the LCP-array. In total, this information uses $k + o(k)$ words (using the non-succinct representation of the word-based LCP-array) or $n+3k+o(n)$ bits (using the succinct representation) of space, and can be computed in $O(k)$ time. Call the resulting method *esa-search* (for Enhanced Suffix Array). We have also seen in Chapter 5 that with a slightly different RMQ-precomputation it is possible to achieve $O(m \log |\Sigma|)$ matching time (Sect. 5.5). Call the resulting method *esa-log*.

We summarize the different search algorithms in the following theorem (see also Tbl. 7.1).

Theorem 7.6 (Pattern Matching in the Word Suffix Array)**.** *The number of word-aligned occurrences of a pattern P of length m in a text T consisting of k words can be found in alphabet-independent $O(m \log k)$ time using the word suffix array, or in $O(m+\log k)$ time with the help of the word-based LCP-array. Text-independent string matching can be done in $O(m|\Sigma|)$ or $O(m \log |\Sigma|)$ time, using another structure of size $O(k/\log k)$ words in addition to the suffix- and LCP-array.* ■

data set	size (MB)	$\|\Sigma\|$	word separators used
English	333	239	LF, SPC, -
XML	282	97	SPC, /, <, >, ”
sources	201	230	LF, SPC, TAB, ., ,, *, [, (, :, +, -
URLs	70	67	LF, /
random	250	2	SPC

data set	number of words	different words	avg. word length
English	67,868,085	1,220,481	5.27
XML	53,167,421	2,257,660	5.60
sources	53,021,263	2,056,864	3.98
URLs	5,563,810	533,809	13.04
random	10,000,001	9,339,339	26.0

Table 7.2: Test-files used for experimental evaluation and their characteristics. In the word separator column, LF stands for “line feed,” SPC for “space,” and TAB for “tabulator.”

7.7 Experimental Results

The aim is to show the practicability of our method. We implemented the word suffix array in C++ (available at `www.bio.ifi.lmu.de/~fischer`). Instead of using a linear time algorithm for the construction of full-text suffix arrays, we opted for the method from Larsson and Sadakane (1999) because it is known to be fastest in practice among those algorithms that work with integer alphabets (Puglisi et al., 2005).[2] We implemented the search strategies bin-naive, bin-improved, bin-lcp and esa-search from Table 7.1.[3] Unfortunately, we could not compare to the other word-based indexes (Inenaga and Takeda, 2006a,b,c) because there are no publicly available implementations.

For bin-improved we chose $c = 1/4$, so the index occupies $1.25k$ memory words (apart from T, which takes n bytes). For the RMQ-preprocessing of the esa-search we used the method from Alstrup et al. (2002) which is fast in practice, while still being relatively space-conscious (about $1.5k$ words; see Sect. 3.8). With the LCP-array and the suffix array this makes a total of $\approx 3.5k$ words. We also performed tests with the “engineered” representation of RMQ-information from Sect. 3.8, thereby lowering the space consumption of esa-search, while leading to an increase in query times by a factor of roughly 2. But as we will see presently that even with Alstrup et al.'s RMQ-method the query time of esa-search is hardly competitive with the other methods, we opted for showing the results based on the fastest RMQ-algorithm.

[2] Apart from the input array, the algorithm from Larsson and Sadakane (1999) needs just one additional array of size k. As this space can be re-used for the final word-based suffix array, this constitutes no extra space.

[3] We did not include the strategy esa-log, as we expect its overhead to pay off only for very large alphabets.

data set	bin-naive	bin-improved	bin-lcp	esa-search	peak 1–3
English	600.2	664.2	859.1	1,296.0	1,118.0
XML	485.2	485.5	688.1	1,024.3	890.9
sources	403.4	403.6	605.6	940.6	807.9
URLs	90.4	90.7	111.6	145.1	132.9
random	286.1	286.3	324.2	385.0	362.4

Table 7.3: Final space consumption (including the text) for the four different search algorithms. The column labeled "peak 1–3" gives the peak space consumption at constructing time for methods bin-naive, bin-improved and bin-lcp. The peak space consumption for esa-search is that of the final index.

We tested our algorithms on the files "English," "XML," and "sources" from the Pizza & Chili-site (Ferragina and Navarro, 2005), some of them truncated, plus one file of URLs from the `.eu` domain.[4] These files cover some typical applications of word-based indexes: natural language, structured data, and prefix- or path-search on hosts/domains in URLs. To test the search algorithms on a small alphabet, we also generated an artificial data set by taking words of random length (uniformly from 20 to 30) and letters uniformly from $\Sigma = \{a, b\}$. See Table 7.2 for the characteristics of the evaluated data sets. All tests were performed on an Athlon XP 3300 with 2GB of RAM under Linux. All programs were compiled with `g++`, using the options "-O3 -fomit-frame-pointer -funroll-loops."

Table 7.3 shows the space consumption for the four different search methods. The first four columns show the space (in MB) of the final index (including the text) for the different search algorithms it can subsequently support. Column labeled "peak 1–3" gives the peak memory usage at construction time for the methods in the first three columns; the peak usage for method esa-search is the same as that of the final index. Concerning the construction time, Tbl. 7.4 shows that most part of the preprocessing time is needed for the construction of the pure word suffix array (method bin-naive); the preprocessing times for the other methods are only slightly longer than that for bin-naive, with bin-improved being the fastest and esa-search being the slowest.

To see the advantage of our method over the naive algorithm which prunes the full-text suffix array to obtain the word suffix array, Table 7.5 shows the construction times and peak space consumption of two state-of-the-art algorithms for constructing (full-text) suffix arrays: MSufSort in its latest version 3.0 (Maniscalco and Puglisi, 2007), and deep-shallow (Manzini and Ferragina, 2004). Note that the figures given in Table 7.5 are pure construction times for the *full-text* suffix array; pruning this is neither included in time nor space. First look at the peak space consumption in Table 7.5. MSufSort needs about $7n$ bytes if the input text cannot be overwritten (it therefore failed for the

[4] Available at `http://law.dsi.unimi.it/index.php`. Last access in January 2007.

data set	bin-naive	bin-improved	bin-lcp	esa-search
English	533.64	558.89	631.57	639.95
XML	328.38	341.95	370.88	377.54
sources	281.12	295.95	323.04	329.74
URLs	45.75	46.60	47.14	47.97
random	224.75	228.17	239.89	241.33

Table 7.4: Preprocessing times for the four different search algorithms.

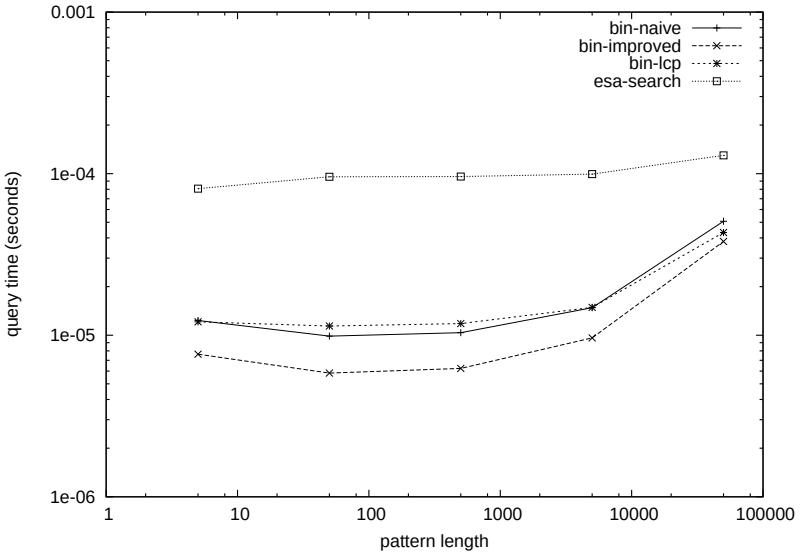
	peak space consumption (MB)		construction time	
data set	MSufSort-3.0	deep-shallow	MSufSort-3.0	deep-shallow
English	—	1,365.3	—	755.8
XML	1,976.9	1,129.7	363.9	410.8
sources	1,407.7	804.4	193.4	260.6
URLs	484.3	276.7	75.2	71.0
random	1,735.6	991.8	452.7	332.2

Table 7.5: Space consumption (including the text) and construction times for two different state-of-the-art methods to construct (full-text) suffix arrays.

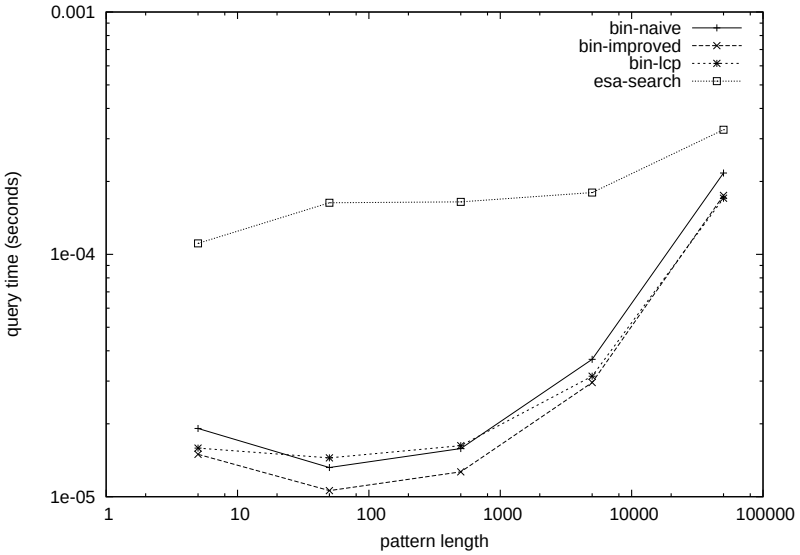
largest data set), and deep-shallow needs about $5n$ bytes. These two columns should be compared with the column labeled "peak 1–3" in Table 7.3, because this column gives the space needed to construct the pure word suffix array (i.e., $12k + n$ bytes in our implementation). For all but one data set our method uses significantly less space than both MSufSort (20.9–57.4%) and deep-shallow (36.5–81.9%). Only for file "sources" space consumption is more or less that of deep-shallow, because it consists of relatively short words compared to the other files; cf. Table 7.2. For the construction time, compare the last two columns in Table 7.5 with the preprocessing time for bin-naive in Table 7.4. Again, our method is almost always fastest (49.6–90.1% and 64.4–80.2% advantage over deep-shallow and MSufSort, respectively); the difference would be even larger if we did include the time needed for pruning the full-text suffix array in order to get the word-based one.

We finally tested the different search strategies themselves. In particular, we posed 300,000 counting queries to each index (i.e., determining the interval of pattern P in A) for patterns of length 5, 50, 500, 5,000, and 50,000. The results can be seen in Fig. 7.4–7.6.[5] We differentiated between *random* patterns (sub-figures (a)) and *occurring* patterns (b). In the former case we searched for arbitrary sub-strings from the input text, and in the latter case we only searched for sub-strings starting at word boundaries. There are several interesting points to note. First, the improved binary search is almost always

[5] We omit the results for the sources- and URLs-data sets as they strongly resemble those for the XML-data set.

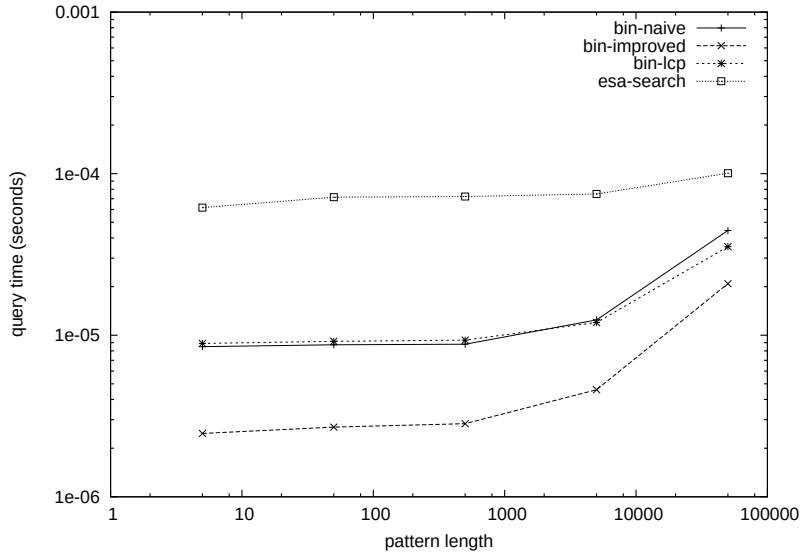


(a) random patterns

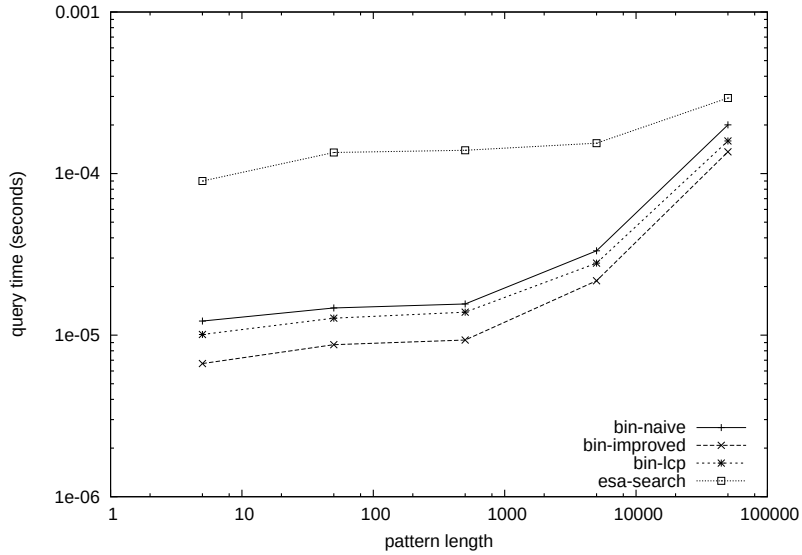


(b) occurring patterns

Figure 7.4: Practical performance of the algorithms from Section 7.6 on the English data set (average over 300,000 counting queries; time for index construction not included). All axes have a logarithmic scale.

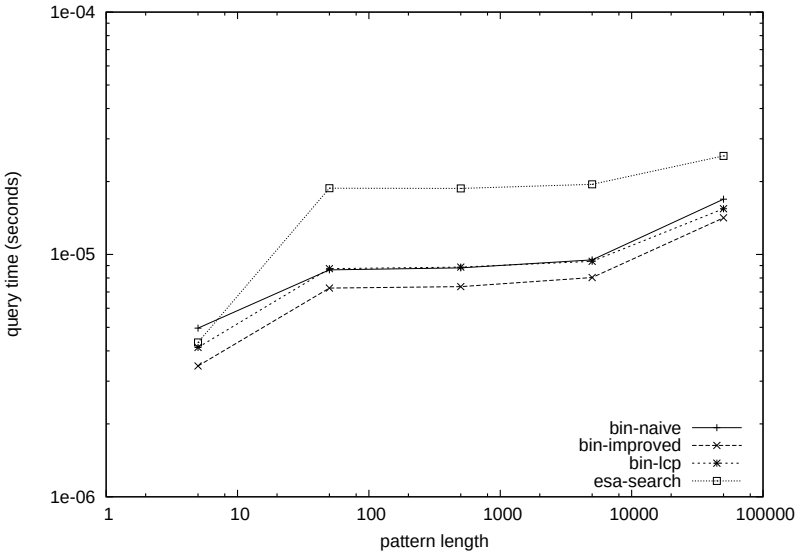


(a) random patterns

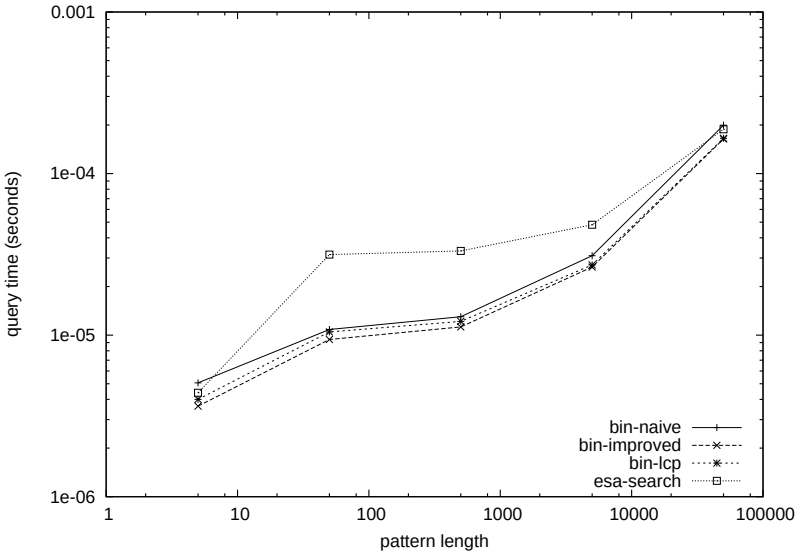


(b) occurring patterns

Figure 7.5: Practical performance of the algorithms from Section 7.6 on the XML data set (average over 300,000 counting queries; time for index construction not included). All axes have a logarithmic scale.



(a) random patterns



(b) occurring patterns

Figure 7.6: Practical performance of the algorithms from Section 7.6 on the random data set (average over 300,000 counting queries; time for index construction not included). All axes have a logarithmic scale.

the fastest; the difference to the other methods is even more significant for the randomly generated patterns. Second, the esa-search is not competitive with the other methods, apart from very long occurring patterns on a very small alphabet (Fig. 7.6 (b)). And third, the query time for the methods based on binary search can actually be higher for short patterns than for long patterns (Fig. 7.4). This is the effect of narrowing down the search for the right border when searching for the left one (cf. Alg. 7.2). Because short patterns tend to occur very often (especially in English texts), searching for the right border starts with a relatively large initial search interval when the patterns are short and thus takes more time.

Finally, to see the impact of the reduced number of binary search steps, we also posed the same queries to a full-text suffix array (without post-processing the results). As matching times increased by a factor of 2–3 for both bin-naive and bin-improved, these curves are not included in the graphs.

7.8 Conclusions

We have seen a space- and time-optimal algorithm to construct suffix arrays on words. The most striking property is the simplicity of our approach, reflected in the good practical performance. This supersedes all the other known approaches based on suffix trees, DAWG and compact DAWG.

As future research issues we point out the following two. In a similar manner as we compressed T (Corollary 7.2), one could compress the word-based suffix array A by probably resorting the ideas on word-based Burrows-Wheeler Transform (Isal and Moffat, 2001) and alphabet-friendly compressed indexes (Ferragina et al., 2007). This would have an impact not only in terms of space occupancy (i.e., better modeling of T, and thus better compression), but also on the search performance of those indexes because they execute $O(1)$ random memory-accesses per searched/scanned character. With a word-based index this could be turned to $O(1)$ random memory-accesses per searched/scanned word, with a significant practical speed-up in the case of very large texts possibly residing on disk.

The second research issue regards the *sparse* string-matching problem in which the set of points to be indexed is given as an arbitrary set, not necessarily coinciding with word boundaries. As pointed out in the introduction, this problem is still open, though being relevant for texts such as biological sequences where natural word boundaries do not occur.

Summary of Notation

notation	meaning
$A[1,n]$	array of n numbers indexed from 1 to n
$A[i,j]$	subarray of $A[1,n]$ containing $A[i], A[i+1], \ldots, A[r]$
$A \circ B$	concatenation of arrays A and B
$[l:r]$	set of integers $\{l, l+1, \ldots, r\}$
$\log n$	binary logarithm of n
$\ln n$	natural logarithm of n
$\log^{[k]} n$	$\log\log\ldots\log n$ (there are k log's)
$\log^* n$	iterated logarithm: $\min\{k : \log^{[k]} \leq 1\}$
$\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots\}$	alphabet containing letters $\mathtt{a}, \mathtt{b}, \ldots$
Σ^k $(\Sigma^\star)$	set of length-k (or all) strings over Σ
$T_{1..n} \in \Sigma^\star$ $(\epsilon \in \Sigma^\star)$	string with symbols $T_1, \ldots, T_n \in \Sigma$ (or empty string)
$T_{i..j}$	substring of T ranging from symbol i to j (or ϵ if $j < i$)
$\lvert S \rvert$	(a) set size, (b) string length if $S \in \Sigma^\star$
$S < T$	$S \in \Sigma$ lexicographically less than $T \in \Sigma$
$S \trianglelefteq T$ $(S \sqsubseteq T)$	S is a substring (or prefix) of T
SA	suffix array for T, defined by $T_{\mathsf{SA}[i-1]..n} < T_{\mathsf{SA}[i]..n}$
SA^{-1}	inverse suffix array, defined by $\mathsf{SA}[\mathsf{SA}^{-1}[i]] = i$
$\textsc{lce}_T(i,j)$	length of longest common prefix of $T_{i..n}$ and $T_{j..n}$
LCP	LCP-array, defined by $\mathsf{LCP}[i] = \textsc{lce}_T(\mathsf{SA}[i], \mathsf{SA}[i-1])$
$H_k(T)$	k'th order empirical entropy of $T \in \Sigma^\star$
$\mathrm{rank}_p(B,i)$	number of occurrences of p in $B[1,i]$
$\mathrm{select}_p(B,i)$	position of i'th occurrence of p in B
T_v	subtree of T rooted at v if T is a rooted tree
$\textsc{rmq}_A(l,r)$	range minimum query: $\arg\min_{i \in [l:r]}\{A[i]\}$
$\langle p(n), q(n) \rangle$	preprocessing scheme for RMQ with construction time $p(n)$ & query time $q(n)$

notation	meaning
$[\![s(n), t(n)]\!]$	preprocessing scheme for RMQ with peak space consumption $s(n)$ & final space $t(n)$
$\mathcal{C}^{\text{can}}(A)$ ($\mathcal{C}_i^{\text{can}}(A)$)	Canonical Cartesian Tree of array A (or $A[1, i]$)
$\mathcal{C}^{\text{ext}}(A)$ ($\mathcal{C}^{\text{sup}}(A)$)	Extended (or Super) Cartesian Tree of array A
C_s ($\hat{C}_s$)	s'th Catalan (or Super Catalan) Number
C_{pq} ($\hat{C}_{pq}$)	Ballot (or Super Ballot) Numbers
$\text{LCA}_T(v, w)$	lowest common ancestor of nodes v and w in tree T
$x \ll k$	x shifted left k bits: $\lfloor 2^k x \rfloor$
OR	bitwise logical "or"
$\ell - [l : r]$	ℓ-interval in LCP from l to r
RMQ^{med}	range-pseudo-median-of-minima-query
$\mathcal{D} \subseteq \Sigma^\star$	(multi-)set of strings over Σ
$\lvert\mathcal{D}\rvert$	number of strings in $\mathcal{D}$
$\lVert\mathcal{D}\rVert$	total length of strings in $\mathcal{D}$: $\sum_{\phi \in \mathcal{D}} \lvert\phi\rvert$
$\mathsf{freq}(\phi, \mathcal{D})$	number of strings in $\mathcal{D}$ containing $\phi \in \Sigma^\star$
$\mathsf{supp}(\phi, \mathcal{D})$	support of ϕ, defined as $\mathsf{freq}(\phi, \mathcal{D})/\lvert\mathcal{D}\rvert$
$\mathsf{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi)$	growth-rate of ϕ, defined as $\mathsf{supp}(\phi, \mathcal{D}_1)/\mathsf{supp}(\phi, \mathcal{D}_2)$

BIBLIOGRAPHY

M. I. Abouelhoda and E. Ohlebusch. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms*, 3(2–4):321–341, 2005.

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Dokl. Akad. Nauk. SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1962, 1962.

P. K. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. Chapman & Hall/CRC, 2nd edition, 2004.

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proc. Int. Conf. on Management of Data*, pages 207–216. ACM Press, 1993.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5(1):115–132, 1976.

S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 258–264. ACM Press, 2002.

S. F. Altschul et al. Basic local alignment search tool. *J. Molecular Biology*, 215:403–410, 1990.

S. Aluru, editor. *Handbook of Computational Molecular Biology*. Chapman & Hall/CRC, 2006.

A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proc. CPM*, volume 4580 of *LNCS*, pages 286–294. Springer, 2007.

A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

Antonitio, P. J. Ryan, W. F. Smyth, A. H. Turpin, and X. Yu. New suffix array algorithms — linear but not fast? In *Proc. Fifteenth Australasian Workshop Combinatorial Algorithms (AWOCA)*, pages 148–156, 2004.

A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer, 1985.

V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk. SSSR*, 194:487–488, 1970. English translation in *Soviet Math. Dokl.*, 11:1209–1210, 1975.

M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

F. Birzele and S. Kramer. A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics*, 22(24):2628–2634, 2006.

A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.

A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34 (3):578–595, 1987.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

S. Chan, B. Kao, C. L. Yip, and M. Tang. Mining emerging substrings. In *Proc. of the Itl. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 119–126. IEEE Computer Society, 2003.

B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proc. Symposium on Computational Geometry*, pages 131–139. ACM Press, 1989.

G. Chen, S. J. Puglisi, and W. F. Smyth. LZ factorization using less time and space. *Mathematics in Computer Science (MCS) Special Issue on Combinatorial Algorithms*, 2007. To appear.

K.-Y. Chen and K.-M. Chao. On the range maximum-sum segment query problem. In *Proc. ISAAC*, volume 3341 of *LNCS*, pages 294–305. Springer, 2004.

D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

L. De Raedt, M. Jäger, S. D. Lee, and H. Mannila. A theory of inductive query answering. In *Proc. ICDM*, pages 123–130. IEEE Computer Society, 2002.

G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. KDD*, pages 43–52. ACM Press, 1999.

M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

P. Ferragina and J. Fischer. Suffix arrays on words. In *Proc. CPM*, volume 4580 of *LNCS*, pages 328–339. Springer, 2007.

P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

P. Ferragina and G. Navarro. The Pizza & Chili Corpus. Available at `http://pizzachili.di.unipi.it`, 2005. Last access in January 2007.

P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007.

P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS*, pages 184–196. IEEE Computer Society, 2005.

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article No. 20, 2007.

J. Fischer and L. De Raedt. Towards optimizing conjunctive inductive queries. In *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, volume 3056 of *LNCS*, pages 625–637. Springer, 2004.

J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. CPM*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006.

J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. Int. Symp. on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.

J. Fischer, V. Heun, and S. Kramer. Fast frequent string mining using suffix arrays. In *Proc. ICDM*, pages 609–612. IEEE Computer Society, 2005.

J. Fischer, V. Heun, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 4213 of *LNCS*, pages 139–150. Springer, 2006.

G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. ICALP*, volume 4596 of *LNCS*, pages 533–545. Springer, 2007.

H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM Press, 1984.

L. Georgiadis and R. E. Tarjan. Finding dominators revisited: extended abstract. In *Proc. SODA*, pages 869–878. ACM/SIAM, 2004.

I. M. Gessel. Super ballot numbers. *J. Symbolic Computation*, 14(2–3):179–194, 1992.

G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 3, pages 66–82. Prentice-Hall, 1992.

R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. CPM*, volume 4009 of *LNCS*, pages 294–305. Springer, 2006.

R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2): 378–407, 2005.

R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. ACM/SIAM, 2003.

D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

D. Gusfield and J. Stoye. Linear time algorithm for finding and representing all tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

W.-K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *Proc. CPM*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002.

W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS*, pages 251–260. IEEE Computer Society, 2003.

L. C. K. Hui. Color set size problem with application to string matching. In *Proc. CPM*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.

S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. CPM*, volume 4009 of *LNCS*, pages 60–71. Springer, 2006a.

S. Inenaga and M. Takeda. Sparse compact directed acyclic word graphs. In *Proc. Prague Stringology Conf. 2006*, pages 197–211, 2006b.

S. Inenaga and M. Takeda. Sparse directed acyclic word graphs. In *Proc. SPIRE*, volume 4209 of *LNCS*, pages 61–73. Springer, 2006c.

S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005.

R. Y. K. Isal and A. Moffat. Word-based block-sorting text compression. In *Proc. Australasian Conf. on Computer Science*, pages 92 – 99. IEEE Press, 2001.

G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA*, pages 575–584. ACM/SIAM, 2007.

J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON*, volume 1090 of *LNCS*, pages 219–230. Springer, 1996.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):1–19, 2006.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

D. K. Kim and H. Park. A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In *Proc. CPM*, volume 3537 of *LNCS*, pages 33–44. Springer, 2005.

D. K. Kim, J. E. Jeon, and H. Park. An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In *Proc. SPIRE*, volume 3246 of *LNCS*, pages 138–149. Springer, 2004.

D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2–4):126–142, 2005.

D. E. Knuth. *The Art of Computer Programming Volume 4 Fascicle 4: Generating All Trees; History of Combinatorial Generation*. Addison-Wesley, 2006.

D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 3rd edition, 1997.

P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2–4):143–156, 2005.

S. Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proc. STOC*, pages 220–230. ACM Press, 1986.

G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. Comput. Biol.*, 8(1):1–18, 2001.

N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Lund, Sweden, 1999.

S. D. Lee and L. De Raedt. An efficient algorithm for mining string databases under constraints. In *Proc. 3rd Itl. Workshop on Knowledge Discovery in Inductive Databases*, volume 3377 of *LNCS*, pages 108–129. Springer, 2005.

W. Ludwig et al. ARB: a software environment for sequence data. *Nucleic Acids Res.*, 32(4):1363–1371, 2004.

M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

V. Mäkinen. Compact suffix array — a space efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003. Special Issue - Computing Patterns in Strings.

V. Mäkinen. *Parameterized Approximate String Matching and Local-Similarity-Based Point-Pattern Matching*. PhD thesis, Department of Computer Science, University of Helsinki, Report A-2003-6, 2003.

V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. Computing*, 12(1):40–66, 2005.

U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

M. A. Maniscalco and S. J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM J. Experimental Algorithmics*, 2007. To appear. Software available at `http://www.michael-maniscalco.com/msufsort.htm`.

H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3): 407–430, 2001.

G. Manzini. Two space saving tricks for linear time lcp array computation. In *Proc. Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004. Software available at `http://www.mfn.unipmn.it/~manzini/lightweight`.

C. Martínez and S. Roura. Randomized binary search trees. *J. ACM*, 45(2): 288–323, 1998.

D. Merlini, R. Sprugnoli, and M. C. Verri. Waiting patterns for a printer. *Discrete Applied Mathematics*, 144(3):359–373, 2004.

J. I. Munro. Tables. In *Proc. Conf. on the Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.

J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. ICALP*, volume 3142 of *LNCS*, pages 1006–1015. Springer, 2004.

J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 345–356. Springer, 2003.

S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666. ACM/SIAM, 2002.

E. W. Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

J. C. Na. Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets. In *Proc. CPM*, volume 3537 of *LNCS*, pages 57–67. Springer, 2005.

G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.

G. Nong and S. Zhang. Optimal lightweight construction of suffix arrays for constant alphabets. In *Proc. Workshop on Algorithms And Data Structures (WADS)*, volume 4619 of *LNCS*, pages 613–624. Springer, 2007.

R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

C. K. Poon and W. K. Yiu. Opportunistic data structures for range queries. In *Proc. COCOON*, volume 3595 of *LNCS*, pages 560–569. Springer, 2005.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. The performance of linear time suffix sorting algorithms. In *Proc. Data Compression Conf. (DCC)*, pages 358–367. IEEE Computer Society, 2005.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article No. 4, 2007.

R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242. ACM/SIAM, 2002.

M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.

K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2007a. doi: 10.1007/s00224-006-1198-x.

K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007b.

K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.

B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proc. CPM*, volume 3109 of *LNCS*, pages 347–358. Springer, 2004.

K.-B. Schürmann and J. Stoye. Counting suffix arrays and strings. In *Proc. SPIRE*, volume 3772 of *LNCS*, pages 55–66. Springer, 2005.

K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

J. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, 1980.

T. Shibuya and I. Kurochkin. Match chaining algorithms for cDNA mapping. In *Proc. Workshop on Algorithms in Bioinformatics (WABI)*, volume 2812 of *LNCS*, pages 462–475. Springer, 2003.

R. P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.

R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, volume 4580 of *LNCS*, pages 205–215. Springer, 2007.

N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen. Engineering a compressed suffix tree implementation. In *Proc. Workshop on Experimental Algorithms (WEA)*, LNCS 4525, pages 217–228. Springer-Verlag, 2007.

J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.

L. Wang, H. Zhao, G. Dong, and J. Li. On the complexity of finding emerging patterns. *Theor. Comput. Sci.*, 335(1):15–27, 2005.

P. Weiner. Linear pattern matching algorithms. In *Proc. Annual Symp. on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

M. Yamamoto and K. W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, 2001.

A. C.-C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record*, 25(3):10–15, 1996.

List of Conference Abbrevations

COCOON	Annual Itl. Computing and Combinatorics Conf.
CPM	Annual Symp. on Combinatorial Pattern Matching
FOCS	Annual Symp. on Foundations of Computer Science
ICALP	Itl. Colloquium on Automata, Languages and Programming
ICDM	IEEE Itl. Conf. on Data Mining
ISAAC	Itl. Symp. on Algorithms and Computation
KDD	ACM SIGKDD Itl. Conf. on Knowledge Discovery & Data Mining
SODA	ACM-SIAM Symp. on Discrete Algorithms
SPIRE	String Processing and Information Retrieval Symp.
STOC	ACM Symp. on Theory of Computing